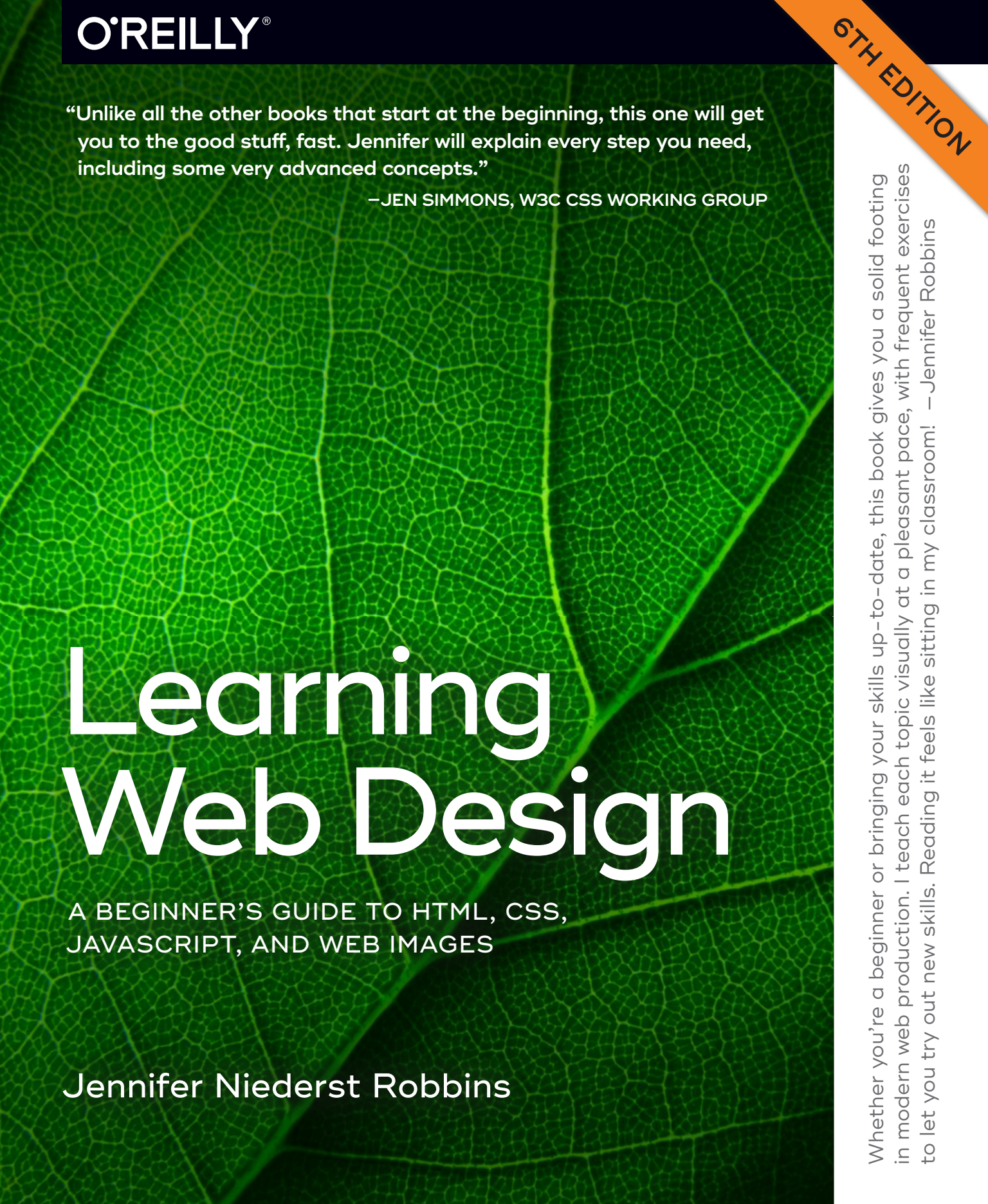

O'REILLY®
6TH EDITION
"Unlike all the other books that start at the beginning, this one will get you to the good stuff, fast. Jennifer will explain every step you need, including some very advanced concepts."
–JEN SIMMONS, W3C CSS WORKING GROUP
Learning Web Design
A BEGINNER'S GUIDE TO HTML, CSS, JAVASCRIPT, AND WEB IMAGES
Jennifer Niederst Robbins
Whether you're a beginner or bringing your skills up-to-date, this book gives you a solid footing in modern web production. I teach each topic visually at a pleasant pace, with frequent exercises to let you try out new skills. Reading it feels like sitting in my classroom! –Jennifer Robbins

Learning Web Design, 6th Edition

A Beginner's Guide to HTML, CSS, JavaScript, and Web Images

by Jennifer Niederst Robbins with contributions by Aaron Gustafson

Do you want to build web pages but have no prior experience? This friendly guide is the perfect place to start. You'll begin at square one, learning how the web and web pages work, then steadily build from there. By the end of the book, you'll have the skills to create a simple site with multicolumn pages that adapt for mobile devices.

Each chapter provides exercises to help you learn various techniques plus short quizzes to make sure you understand key concepts.

This thoroughly revised sixth edition, featuring a rewritten and expanded JavaScript section, is ideal for students and professionals of all backgrounds and skill levels. It's simple and clear enough for beginners yet thorough enough to be a useful reference for experienced developers keeping their skills up-to-date.

- **Build HTML pages with text, links, images, tables, and forms**
- **Use style sheets (CSS) for formatting text, colors, backgrounds, multicolumn page layout, and even simple animation**
- **Use CSS Flexbox and Grid Layout for sophisticated and flexible page designs**
- **Learn the ins and outs of responsive web design to make web pages look great on all devices**
- **Learn basic JavaScript syntax and use it to add functionality to web pages**
- **Create and optimize web images so they'll download as quickly as possible**

ABOUT THE AUTHOR

Jennifer Niederst Robbins designed the first commercial website, O'Reilly's Global Network Navigator (GNN), in 1993. Author of the best-selling *Web Design in a Nutshell*, she taught web design at the Massachusetts College of Art in Boston and was one of the cofounders of the Artifact Conference, focused on mobile web design.

US $79.99 CAN $99.99
ISBN: 978-1-098-13768-7

O'REILLY®

Sixth Edition

LEARNING WEB DESIGN

A BEGINNER'S GUIDE TO HTML, CSS, JAVASCRIPT, AND WEB IMAGES

Jennifer Niederst Robbins

with contributions by Aaron Gustafson

O'REILLY®

Learning Web Design, Sixth Edition

A Beginner's Guide to HTML, CSS, JavaScript, and Web Images

by Jennifer Niederst Robbins
with contributions by Aaron Gustafson

Copyright © 2025 Littlechair, Inc., O'Reilly Media, Inc. All rights reserved.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.
O'Reilly Media books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

ACQUISITIONS EDITOR: David Michelson

DEVELOPMENT EDITOR: Jeff Bleiel

PRODUCTION EDITOR: Jonathon Owen

COVER DESIGNER: Edie Freedman

INTERIOR DESIGNER: Jennifer Robbins

PRINT HISTORY:

March 2001:	First edition.
June 2003:	Second edition.
June 2007:	Third edition.
August 2012:	Fourth edition.
May 2018:	Fifth edition.
June 2025:	Sixth edition.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Learning Web Design*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

ISBN: 978-1-098-13768-7
[LSI] [2025-06-18]

CONTENTS

FOREWORD xi
PREFACE xiii

PART I. GETTING STARTED

1. Teams and Tools 3
Getting Started in Web Design 4
Roles and Responsibilities 4
Tools of the Trade 15
What You've Learned 21
Test Yourself 22

2. How the Web Works 23
The Internet Versus the Web 23
Serving Up Your Information 24
A Word About Browsers 25
Web Page Addresses (URLs) 26
The Anatomy of a Web Page 30
Putting It All Together 34
Test Yourself 36

3. Some Big Concepts You Need to Know 37
A Multitude of Devices 38
Sticking with the Standards 39
Progressive Enhancement 40
Responsive Web Design 41
Accessibility 43
Site Performance 45
Test Yourself 48

PART II. HTML for Structure

4. Creating a Simple Page 51

A Web Page, Step-by-Step 51
Launch a Text Editor 52
Step 1: Start with Content 55
Step 2: Give the HTML Document Structure 57
Step 3: Identify Elements 61
Step 4: Add an Image 64
Step 5: Change the Look with a Style Sheet 68
When Good Pages Go Bad 69
Validating Your Documents 71
HTML Resources 72
Test Yourself 73
Element Review: HTML Document Setup 74

5. Marking Up Text 75

Paragraphs 75
Headings 76
Thematic Breaks (Horizontal Rules) 78
Lists 78
More Content Elements 82
Organizing Page Content 85
The Text-Level Element Roundup 90
Generic Elements (div and span) 101
Improving Accessibility with ARIA 105
Character Escapes 107
Putting It All Together 110
Test Yourself 113
Element Review: Text Elements 114

6. Adding Links 115

The href Attribute 116
Linking to Pages on the Web 117
Linking Within Your Own Site 118
Targeting a New Browser Window 128
Mail Links 129
Telephone Links 130
Test Yourself 131
Element Review: Links 132

7. Adding Images 133
First, a Word on Image Formats 134
The img Element 135
Adding SVG Images 141
The picture Element for Alternative Image Formats 148
A Quick Introduction to Responsive Images 149
And We're Finished! 150
Test Yourself 150
Element Review: Images 151

8. Table Markup 153
How to Use Tables 153
Minimal Table Structure 155
Table Headers 157
Spanning Cells 158
Table Accessibility 159
Row and Column Groups 161
Wrapping Up Tables 163
Test Yourself 165
Element Review: Tables 166

9. Forms 167
How Forms Work 167
The form Element 169
Variables and Content 172
The Great Form Control Roundup 173
Form Accessibility Features 191
Form Layout and Design 196
Test Yourself 198
Element Review: Forms 199

10. Embedded Media 205
Window-in-a-Window (iframe) 205
Multipurpose Embedder (object) 208
Video and Audio 208
Canvas 217
Test Yourself 220
Element Review: Embedded Media 221

PART III. CSS FOR PRESENTATION

11. Introducing Cascading Style Sheets 225

The Benefits of CSS 225
How Style Sheets Work 226
The Big Concepts 232
CSS Units of Measurement 241
Developer Tools Right in Your Browser 244
Moving Forward with CSS 246
Test Yourself 247

12. Formatting Text 249

Basic Font Properties 249
Descendant, ID, and Class Selectors 268
Specificity 101 272
Text Line Adjustments 275
Underlines and Other "Decorations" 278
Changing Capitalization 279
Spaced Out 280
Text Shadow 281
Changing List Bullets and Numbers 284
Finishing Up Text Styles 287
Test Yourself 288
CSS Review: Font and Text Properties 290

13. Colors and Backgrounds 293

Color Spaces and Models 293
Specifying Color Values 296
Foreground Color 306
Background Color 307
Playing with Opacity 309
Background Images 311
Like a Rainbow (Gradients) 321
Wrapping It Up 326
Test Yourself 326
CSS Review: Color and Background Properties 328

14. Selectors 329

Selectors You've Learned So Far 329
Attribute Selectors 330

Pseudo-Class Selectors 332
Pseudo-Element Selectors 348
External Style Sheets 350
Wrapping It Up 354
Test Yourself 355

15. Thinking Inside the Box 359
The Element Box 359
Specifying Box Dimensions 360
Padding 366
Borders 371
Margins 381
Assigning Display Types 387
Box Drop Shadows 389
Test Yourself 391
CSS Review: Box Properties 392

16. Floating and Positioning 395
Normal Flow 395
Floating 396
Fancy Text Wrap with CSS Shapes 407
Positioning Basics 411
Relative Positioning 413
Absolute Positioning 414
Fixed Positioning 421
Stacking Order 422
Test Yourself 424
CSS Review: Floating and Positioning Properties 425

17. Getting in Line with Flexbox 427
Setting Up a Flex Container 428
Controlling the "Flow" Within the Container 430
Controlling the Alignment of Flex Items 434
Flexbox Alignment Techniques 441
Determining How Items "Flex" in the Container 445
Changing the Order of Flex Items 450
Flexbox for Page Layout 453
More Flexbox Resources 454
Test Yourself 456
CSS Review: Flexbox Properties 457

18. Grid Layout 459
Grid Layout Overview 460
Implicit Grids and Default Grid Behavior 463
Explicit Grids 469
Placing Items on the Grid 477
The grid Shortcut Property 487
Grid Alignment and Spacing 489
Responsive Grids 494
Additional Resources 501
Test Yourself 502
CSS Review: Layout Properties 504

19. Responsive Web Design 505
Why RWD? 505
The Responsive Recipe 507
Setting the Viewport 507
Fluid Layouts 509
Flexible Images 510
Query Magic 512
Designing Responsively 529
A Few Words About Testing 546
More RWD Resources 548
Test Yourself 549

20. Transitions, Transforms, and Animation 551
Ease-y Does It (CSS Transitions) 551
CSS Transforms 561
Keyframe Animations 571
Wrapping Up 576
Test Yourself 576
CSS Review: Transitions, Transforms, and Animation 578

21. More CSS Techniques 579
Styling Tips for Forms 579
A Clean Slate (Reset and Normalize) 583
Supporting Older Browsers (CSS Feature Detection) 587
CSS Custom Properties (Variables) 592
A Quick Introduction to CSS Nesting 596
An Even Quicker Introduction to CSS Layers 598
CSS Frameworks 600
Wrapping Up Style Sheets 602

Test Yourself ... 602
CSS Review ... 604

PART IV. JavaScript for Behavior

22. Getting Started with JavaScript ... 607
Meet a Script ... 608
Statements ... 612
Numbers and Math ... 613
Strings ... 618
Variables ... 621
Embedded Versus External JavaScript ... 627
The JavaScript Console ... 628
Wrapping Up ... 631
Test Yourself ... 633

23. Functions, Conditionals, and Loops ... 635
Functions ... 636
Functions Can Receive and Return Information ... 639
Conditional Statements ... 645
Arrays ... 652
Loops Make Repetitive Tasks Easier ... 654
Wrapping Up ... 659
Test Yourself ... 660

24. Working with the DOM and Events ... 663
UI as a Progressive Enhancement ... 664
Generating New Markup ... 665
JavaScript Events ... 678
Refactoring for Accessibility ... 684
Wrapping Up ... 689
Test Yourself ... 690

25. Next-Level JavaScript ... 693
JavaScript Libraries ... 694
Web Components ... 698
JavaScript Frameworks ... 701
JavaScript Beyond the Browser ... 711
Progressive Web Apps ... 714
Wrapping Up ... 720
Test Yourself ... 721

PART V. WEB IMAGES

26. Web Image Basics 725

Image Sources 726
Meet the Image Formats 730
Resolution and Reference Pixels 752
Working with Transparency 756
Favicons and Site Icons 762
That's a Wrap on Image Basics! 766
Test Yourself 767

27. Image Production Techniques 769

Image Production Strategy 769
Optimizing Images 773
Responsive Image Sets 785
Tools, Tips, and Techniques 789
Automate It! 794
Wrapping Up Image Production 796
Test Yourself 796

28. SVG 799

Drawing with XML 801
Features of SVG as XML 813
SVG Design Tools 820
SVG Production Tips 822
Scaling SVGs 826
Further SVG Exploration 837
Test Yourself 837
And...WE'RE DONE! 838

PART VI. APPENDICES

A. Answers 841

B. HTML Global Attributes 857

C. CSS Selectors, Level 4 861

D. Shortened URL Reference 867

INDEX 873

FOREWORD

BY BRAD FROST

It was a warm September night somewhere outside of Nashville. The campfire crackled, and because I can't help myself, I plucked away at my acoustic guitar. Based on the laughter emanating from our circle and into the woods, you wouldn't know that the people sitting around the fire had just met each other a few days earlier. Conversation about life, design, music, and technology blurred together, and inside jokes were formed in real time...but even the best of nights has to end, and we knew we had important work to do in the morning.

As we trudged back from the fire back to our Airbnb, I asked Jen—whom I'd been laughing along with all night—the boring question most people lead with when they meet someone: "So what do you do for work?"

Jen casually replied, "Oh, I've written a couple of books about web design."

"Oh, nice! What are they?"

"*Learning Web Design*, *Web Design in a Nutshell*, and a couple others."

I stopped dead in my tracks.

My eyebrows shot up as my neurons connected the dots: the warm and hilarious person I'd been hanging out with also happened to be the author of the very first book I'd ever read about web design. That book, *Web Design in a Nutshell*, ignited in me a passion for the web and set me on a trajectory in life that somehow led to me to being invited to a post-conference retreat with many of my web design heroes. That retreat then triggered a snowball of events for me that has led to a long career full of success, notoriety, friendships, and an enduring passion for the web.

Jennifer Robbins changed my life. Her book changed my life. The web changed my life.

And now, dear reader, you are also reading a book about web design by Jennifer Robbins. I hope Jen's book changes your life, too. Of course, there's no guarantee that it will—that part is up to you—but know that these pages contain the foundational information you need to make incredible things for the World Wide Web.

I've now been a professional web designer for 20 years—a full half of my life!—and I've learned that there are very few absolute truths in our field. But there are four things about this revolutionary medium that I believe to be true:

1. The incredible ideals of the web remain intact: it is a universal, open, and decentralized space where everyone can connect, create, and share knowledge in order to shape a better world.
2. The web is always changing.
3. Learning the fundamentals of the web will provide you with a solid foundation that will outlast many trends, technologies, techniques, and tools.
4. Making things for the web is fun, empowering, and incredibly powerful.

In many respects, the web has changed so much since I first read Jen's book in the early 2000s. But in many respects, it has not. Technologies have advanced, and we create increasingly sophisticated (and often overcomplicated!) digital products using those technologies. But irrespective of the complex systems and abstractions used to create websites and apps these days, the web still comes alive to its users in the form of trusty old HTML, CSS, and JavaScript.

These three languages come together to make the web work. In addition to being increasingly powerful, these three languages are durable and resilient. These three languages help us enshrine and disseminate human knowledge and utility that can help people learn, grow, get things done, connect, and collaborate. These three languages are beautiful.

Unfortunately, entire generations of designers and developers have skipped right over learning these foundational languages in favor of jumping right into flashy frameworks that undoubtedly look good on a résumé. However, today's new hotness is tomorrow's forgotten trend. Frameworks and buzzwords will come and go, but HTML, CSS, and JavaScript will endure for as long as the web exists. Learning these fundamentals will help you—and whatever you create on the web—endure in an ever-changing technology landscape.

With this book, not only will you learn core fundamentals about how to make things for the web, but Jennifer will deliver them to you with warmth and clarity. You'll soon find that it's like your cool friend Jen is sitting beside you, clearly and patiently showing you neat stuff you can do with the web. While this book is comprehensive, it's not daunting thanks to Jennifer's friendly nature and clear writing style.

You're in for a treat. Jennifer's books about web design changed my life for the better, and this book can do the same for you too—if you allow it to. So, read this book. Make things for the web. Share with the world. Repeat.

I'm so excited for you, and I can't wait to see what you'll create!

—Brad Frost
Web designer, consultant, author of *Atomic Design*

PREFACE

Look at you...reading the Preface! I'm impressed by your dedication! Hello, and welcome to the *sixth* edition of *Learning Web Design*.

It's hard to believe that I wrote the first edition of this book 25 years ago. To say things have changed since then would be a vast understatement. Web design and development have gotten so much better! We have more and better tools at our disposal for layout, performance, interactivity, and efficiency than we did then. In fact, the development landscape has improved significantly even since the publication of the fifth edition in 2018. The inevitable byproduct of that evolution, however, is that it has become more complicated, and there is a lot more to learn.

If that feels overwhelming, don't worry. Some say my superpower is taking big, gnarly, complicated concepts and making them clear and understandable. This book assumes you are starting from square one, with little or no experience with code. It begins with an explanation of how the web works, and progresses gradually, with exercises and quizzes in each chapter so you can apply and measure what you've learned along the way. By the end of the book, you will have a solid foundation in HTML, CSS, JavaScript, and image production that will serve you well as you learn and practice more on your own. Although I start as simply as possible, I'll also introduce you to advanced features along the way, so you'll have a good overview of the possibilities when you're ready to explore them.

Like the previous ones, this edition addresses the specific needs and concerns of beginners of all backgrounds, including seasoned graphic designers, programmers looking to expand their skills, and anyone else wanting to learn how to make websites. Whether you are reading this book on your own or using it as a companion to a web design course, I hope that it gives you a good head start, and that you have fun in the process.

WHAT'S NEW IN THIS EDITION

Thankfully, web technologies have settled down a bit in recent years, and responsive web design and features that were experimental in HTML5 are now the default ways of doing things. That said, there are still plenty of changes in this sixth edition. Here's an overview.

More Thorough JavaScript Instruction

I'm most excited about the expanded JavaScript section, written by my friend and fellow web old-timer Aaron Gustafson. In addition to being a JavaScript expert, Aaron is a strong advocate for progressive enhancement and accessibility, and he's been writing about the web for decades—including his own book, *Adaptive Web Design: Crafting Rich Experiences with Progressive Enhancement* (***aaron-gustafson.com***). He also wrote the JavaScript chapters in one of my early books, *Web Design in a Nutshell* (published in 2001), so Aaron and I have been collaborating on this sort of thing for a while!

Aaron will lead you through lessons and exercises that will give you a good feel for what it's like to add interactivity to a web page with JavaScript. You will learn the ins and outs of variables, conditionals, loops, and arrays and how to use them to manipulate web content and its presentation. His chapters are a good first step on your JavaScript journey (should you choose to take it!).

Updates Everywhere!

All other chapters have been updated to reflect the latest standards and practices.

The majority of the changes are in the CSS chapters, where the specs are always evolving. There is a new chapter, **Chapter 14, Selectors**, where you'll get a chance to try out newer ways to target page elements for styling. Flexbox and Grid Layout now each have their own chapters. **Chapter 19, Responsive Web Design**, has been expanded with information about design systems and container queries. Finally, **Chapter 20, More CSS Techniques**, includes details on cutting-edge CSS syntax options such as variables, nesting, and layers.

In addition, the image production chapters have had a major overhaul to reflect the requirements of responsive layouts and to keep up with the available image editing and design tools.

Online Articles

To make room for all the new material, some content from the previous edition has been moved to online articles, available at ***learningwebdesign.com/articles/***. You'll find information on responsive image markup, background

images, border images, the command line, CSS Shapes, Git and GitHub, and more. There are references to specific articles throughout the book.

URL Shortening

Because it isn't possible to include everything there is to know about web development in one book, I provide plenty of references to online resources for further learning. Unfortunately, some of those URLs can be really long. In this edition, I have employed O'Reilly's URL shortener to make the most unwieldy URLs easier to type in your browser. For example, instead of typing:

> *developer.mozilla.org/Apps/Fundamentals/Audio_and_video_delivery/Adding_captions_and_subtitles_to_HTML5_video*

you can simply type:

> *oreil.ly/IsL_9*

Better, right? **Appendix D, Shortened URL Reference**, lists the full URL for every shortened version in the book.

HOW THIS BOOK IS ORGANIZED

The book is divided into five parts, each dealing with an important aspect of web development:

Part I, Getting Started

Part I lays a foundation for everything that follows in the book. I start off with some important general information about the web design environment, including the various roles you might play, the technologies you might learn, and the tools that are available to you. You'll get your feet wet right away with HTML and CSS and learn how the web and web pages generally work. I'll also introduce you to some Big Concepts that will get you thinking in the same way that modern web designers think about their craft.

Part II, HTML for Structure

The chapters in **Part II** cover the nitty-gritty of the elements and attributes available to give content semantic structure. We'll cover the markup for text, links, images, tables, forms, and embedded media.

Part III, CSS for Presentation

In the course of **Part III**, you'll progress from learning the basics of Cascading Style Sheets for changing the presentation of text, to creating multicolumn layouts and even adding time-based animation and interactivity to the page. This part of the book provides an introduction to responsive web design as well as techniques that are part of the modern developer's workflow.

Part IV, JavaScript for Behavior

Aaron Gustafson starts **Part IV** with a rundown of JavaScript syntax, so that you can tell a variable from a function. In **Chapter 24, Working with the DOM and Events**, you'll use JavaScript with the DOM to add a theme toggle button to a page. He'll also give you the big picture on how JavaScript is used in modern development, including introductions to libraries and frameworks.

Part V, Web Images

Part V introduces the various image file formats that are appropriate for the web, provides strategies for choosing them as part of a responsive workflow, and describes how to optimize them to make the file sizes as small as possible. It also includes a chapter on SVG graphics, which offer great advantages for responsive layouts and interactivity.

Part VI: Appendices

Part VI holds reference material such as answers to the quiz questions at the end of each chapter, lists of HTML global attributes and CSS selectors, and the list of expanded URLs in the book.

TYPOGRAPHICAL CONVENTIONS

Italic

Used to indicate filenames and directory names, as well as for emphasis

Colored italic

Used to indicate URLs and email addresses

Colored roman

Used for special terms that are being defined

`Constant width`

Used to indicate code examples and keyboard commands

`Colored constant width`

Used for emphasis in code examples

`Constant width italic`

Used to indicate placeholders for attribute and style sheet property values

→

Indicates that a line of code was broken in the text but should remain together on one line in use

Acknowledgments

Once again, many smart and lovely people had my back on this edition.

First, I want to thank Aaron Gustafson for agreeing to come on board as the author of the chapters in **Part IV** and for his patience and grace throughout the process. I'm confident the new chapters will give readers an excellent first experience with writing code.

I want to say a special thanks to my tech reviewers, Brad Frost, Jens Oliver Meiert, and Marija Zaric, whose expertise and attention to detail made this a much better book. Thank you for sprinkling nice comments in there as well.

A second thank you goes to Brad Frost for not only finding a way to fit writing a Foreword into his bonkers schedule, but really knocking it out of the park. Brad is an inspiration professionally, but also as a model for how to live life.

I want to thank my terrific team of folks at O'Reilly Media: Jeff Bleiel (developmental editor), Jonathon Owen (production editor), Rachel Head (copyeditor), Sonia Saruba (proofreader), Potomac Indexing, LLC (indexer), Kristen Brown (production support), and Catherine Dullea (figure review). Special thanks go to InDesign and book production expert Ron Bilodeau, for getting me back up to speed with InDesign and for being a real joy to work with.

Warm thanks also go to Edie Freedman for hiring me as a book designer at O'Reilly 33 years ago and setting my whole life on a new trajectory that most unexpectedly resulted in me becoming the author of 14 books. You are a mentor and a great friend.

Finally, no acknowledgments would be complete without profound appreciation for the love and support of my dearest ones, Jeff and Arlo.

I dedicate this edition to the memory of Pableaux Johnson, my larger-than-life friend, gone too soon.

ABOUT THE AUTHOR

Jennifer Robbins began designing for the web in 1993 as the graphic designer for Global Network Navigator, the first commercial website. In addition to this book, she has written multiple editions of *Web Design in a Nutshell* and *HTML5 Pocket Reference*, published by O'Reilly. She is a founder and organizer of the Artifact Conference, which addressed issues related to mobile web design. Jennifer has spoken at many conferences and has taught beginning web design at Johnson and Wales University in Providence, Rhode Island. When not on the clock, Jennifer enjoys making things, sewing, travel, cooking, and road trips to visit friends all around the country.

O'REILLY ONLINE LEARNING

For more than 40 years, O'Reilly Media has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit ***oreilly.com***.

HOW TO CONTACT US

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)

support@oreilly.com

oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at ***oreil.ly/learningWebDesign_6e***.

For more news and information about our books and courses, see our website at ***oreilly.com***.

Find us on LinkedIn: ***linkedin.com/company/oreilly-media***

Watch us on YouTube: ***youtube.com/oreillymedia***

1

GETTING STARTED

CHAPTER 1

TEAMS AND TOOLS

WORKING IN WEB DESIGN

The web has been around for more than 30 years now, experiencing euphoric early expansion, an economic-driven bust, an innovation-driven rebirth, and constant evolution along the way. One thing is certain: the web as a communication and commercial medium is here to stay.

This book is a good place to start learning what makes the web tick. My goal is to give you a solid foundation in writing code and producing images according to best practices. I'll also provide important context on *why* we do things the way we do. We'll get to writing code very soon, but first, let's take a look at the web design profession.

Maybe you are reading this book as part of a full course on web design and development. Maybe you bought it to expand your current skill set, to open up job possibilities. Whatever your motivation, it will help to be familiar with the many roles in the web design and development field, so you can consider which role is right for you. You don't have to learn to do it *all* (unless you want to!).

In this chapter, we'll look at a variety of jobs that are essential to the making of websites:

- Designers, who are responsible for making the site easy to use and great looking
- Developers, who write the code for the pages themselves as well as the programs that make things work
- Content managers and information architects, who focus on the content of the site and how it is generated, managed, and organized

I'll also introduce the equipment and software commonly used for web-related tasks—in other words, the tools of the trade.

IN THIS CHAPTER

Design specialties

Frontend development

Backend development

Content-related disciplines

Recommended equipment

Web-related software

Share What You Learn

The web design community has thrived on the sharing of information. For many, the author of this book included, sharing what we learn, as we learn it, has led to new roles and opportunities. Even as a beginner, there are people out there who will benefit from your perspective on what you've learned. Sharing is a contribution to the community.

GETTING STARTED IN WEB DESIGN

There are many ways to get involved in web design, from building a small site for yourself to making it a full-blown career. You may enjoy being a "full-stack" web developer or just specializing in one skill. There are a lot of ways you can go.

If you are interested in pursuing web design or production as a career, you'll need to bring your skills up to a professional level. Employers may not require a web design degree or certification, but they will expect to see a portfolio of working sample sites that demonstrate your skills and experience. These might be sites you've created for class assignments, on a volunteer basis, as personal projects, or for a small business or other organization. What's important is that they look professional and have well-written, clean HTML, style sheets, and scripts that show what you can do.

If your involvement is at a smaller scale—say, you just have a site or two you'd like to publish—you may find using a template on an online website service gives you a great head start (see the sidebar **"I Just Want My Own Site"**). Most allow you to tweak the underlying code, so what you learn in this book will help you customize the template to your liking.

I Just Want My Own Site

You don't necessarily need to become a web designer or developer to start publishing on the web. There are many website hosting services that provide templates and drag-and-drop interfaces that make it easy to build a site without knowing how to code. They can be used for anything from full-service ecommerce solutions to small, personal sites (although some services are better suited to one or the other).

Here are a few of the most popular site-building services as of this writing:

- WordPress (*wordpress.com*)
- Squarespace (*squarespace.com*)
- Wix (*wix.com*)

There are many similar services available, some using AI to generate content and designs, so it's worth searching the web to find the one that's right for you.

ROLES AND RESPONSIBILITIES

When I look at a site, I see the multitude of decisions and areas of expertise that went into building it. Sites usually begin with a business plan or other defined mission, ideally backed up with careful research. All of the content must be created and organized. Designers plan and design the overall structure of the site, as well as every page type and each component they contain, keeping in mind the user's experience moving through it and a consistent presentation of the organization's brand identity. And of course, the code must be written and coordinated with what's happening on the server to bring it all to fruition.

Big, well-known sites are created by teams of dozens, hundreds, or even thousands of contributors. There are also sites that are created and maintained by much smaller teams, with just a handful of members. In fact, it's absolutely possible to create a respectable site with a team of only yourself. That's the beauty of the web.

You may end up specializing in just one area of expertise, or working as part of a team of specialists. If you are designing sites on your own, you will need to wear many hats. Consider that the day-to-day upkeep of your household requires you to be part-time chef, housecleaner, accountant, diplomat, gardener, and construction worker—but to you, it's just the stuff you do around the house. As a solo designer, you'll handle many web-related disciplines, but it will just feel like the stuff you do to make a website. In this section, I'll

introduce you to the various disciplines that contribute to the creation of a site, including roles related to design, code, and content.

All Manner of Design

Ah, *design*! It sounds fairly straightforward, but even this simple requirement has been divided into a number of specializations. We'll take a look at a few of the tasks related to designing a site here, but bear in mind that the disciplines often overlap and that the person calling themselves the "designer" often is responsible for more than one (or perhaps even all) of these areas.

User experience and interaction design

Commonly, when people think of design, they think about how something looks. On the web, the first matter of business is designing how the site *works*. Before you pick colors and fonts, it's important to identify the site's goals, how it will be used, and how visitors will move through it. These tasks fall under the disciplines of **user experience** (**UX**) **design** and **interaction design** (**IxD**). There's a lot of overlap between these responsibilities, and it's not uncommon for one person or team to handle all of them.

The user experience designer takes a holistic view of the design process, ensuring the entire experience with the site is favorable. UX design is based on a solid understanding of users and their needs, derived through observations and interviews, balanced with business goals. According to Donald Norman (who coined the term), UX design includes "all aspects of the user's interaction with the product: how it is perceived, learned, and used." For a website or application, that includes the visual design, the user interface, the quality and message of the content, and even how quickly the pages load. The experience must be in line with the organization's brand and business goals in order to be successful.

The goal of the **interaction designer** is to make the site as easy, efficient, and delightful to use as possible. An interaction designer makes decisions about the arrangement of components (content areas, search fields, buttons, calls to action, etc.) on each page type. They also think about pathways through the site and the steps and feedback in processes such as making a purchase or signing up for a service. They make sure the site makes it clear what can be done and how to do it.

The following are deliverables that UX and interaction designers produce:

User research and testing reports

Understanding the needs, desires, and limitations of users is central to the success of the design of the site or web application. The approach of designing around the user's needs is referred to as **user-centered design** (**UCD**), and it is central to contemporary web design. Site designs often begin with user research, including interviews and observations, in order

Do Designers Need to Learn to Code?

In short, yes. Familiarity with HTML and CSS, as well as a general understanding of how content behaves in browsers, is now a requirement of anybody joining a web design team. You may not be responsible for creating the final production code for the site, but as HTML and CSS are the native languages of your medium, you need to know your way around them. Some designers also learn JavaScript, but others draw the line there and let a developer handle the programming.

Code has become central to the visual designer's workflow. Whereas once Photoshop was all you needed to mock up web page designs to send to production, mockups fall short for communicating how designs will look and behave across a wide range of devices and screen sizes. For that reason, working prototypes are preferred and often fall under the designer's responsibility. Fortunately, UI design tools like Figma have been designed to communicate the flexible nature of web pages (they can even generate code for you). It's not uncommon for designers to work out their designs in code (HTML and CSS), or to start the sketching process in an image editor but move to code very quickly to refine the details.

to gain a better understanding of how the site can solve problems or how it will be used. The site's design should also be **inclusive**; that is, it should not alienate any segment of the population who may visit it.

Designers may do a round of user testing at each phase of the design process to ensure the usability of their designs. If users are having a hard time figuring out where to find content or how to move to the next step in a process, then it's back to the drawing board.

Wireframe diagrams

A wireframe diagram shows the structure of a web page using only outlines for each content type and widget (FIGURE 1–1). The purpose of a wireframe diagram is to indicate how the screen real estate is divided and where functionality and content such as navigation, search boxes, form elements, and so on are placed. Colors, fonts, and other visual identity elements are deliberately omitted so as not to distract from the structure of the page. These diagrams are often annotated with instructions for how things should work, so the development team knows what to build. Wireframes may be created in drawing programs or they might be working prototypes designed with code, which have the advantage of communicating basic interactivity.

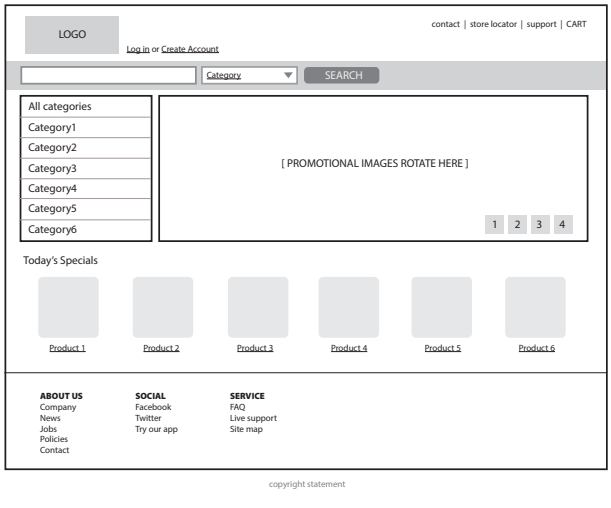


FIGURE 1–1. A wireframe diagram

Site diagram

A site diagram indicates the structure of the site as a whole and how individual pages relate to one another. FIGURE 1–2 shows a very simple site diagram. Some site diagrams fill the walls of entire hallways!

Storyboards and user flow charts

A storyboard traces the path through a site or application from the point of view of a typical user (a **persona** in UX lingo). It usually includes a

script and "scenes" consisting of screen views or the user interacting with the screen. Storyboards aim to demonstrate the steps required to accomplish tasks; they outline possible options and introduce some standard page types. FIGURE 1-3 shows a simple storyboard. A user flow chart is another method for showing how the parts of a site or application are connected, but it tends to focus on technical details rather than telling a story: for example, "when the user does *this*, it triggers *that* function on the server." User flow charts are helpful for documenting the steps in a process, such as member registration or online payments.

There are many books on UX and interaction design, but these are a few of the classics to get you started:

- *About Face: The Essentials of Interaction Design*, 4th edition, by Alan Cooper, Robert Reimann, David Cronin, and Christopher Noessel (Wiley)
- *Designing Interfaces: Patterns for Effective Interaction Design*, 3rd edition, by Jenifer Tidwell, Charles Brewer, and Aynne Valencia (O'Reilly)
- *100 Things Every Designer Needs to Know About People*, 2nd edition, by Susan Weinschenk (New Riders)
- *Designing User Experience: A Guide to HCI, UX and Interaction Design*, 4th edition, by David Benyon (Pearson)
- *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability*, 3rd edition, by Steve Krug (New Riders)
- *The Design of Everyday Things* by Don Norman (Basic Books)
- *The Elements of User Experience: User-Centered Design for the Web and Beyond*, 2nd edition, by Jesse James Garrett (New Riders)

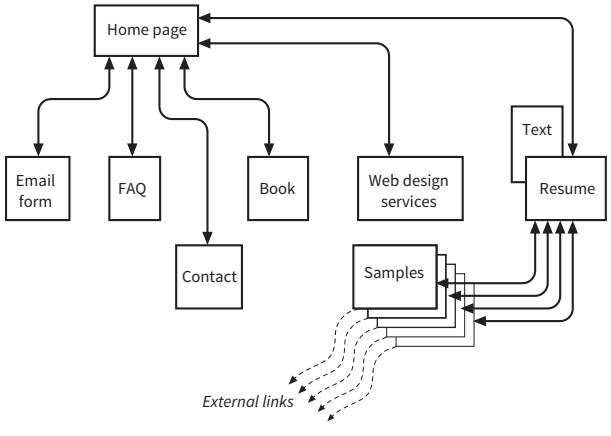


FIGURE 1-2. A simple site diagram

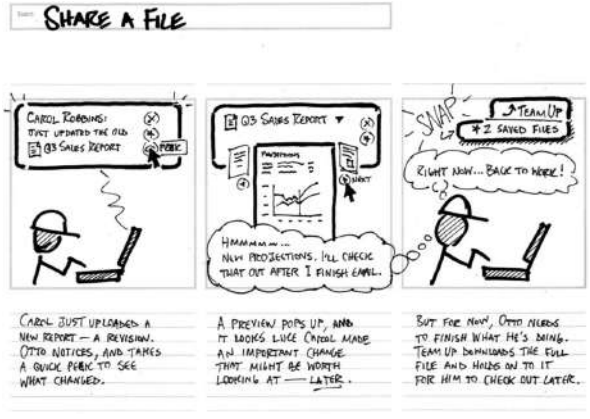

SHARE A FILE
CAROL ROBBINS: JUST UPDATED THE OLD Q3 SALES REPORT
PUBLIC
Q3 SALES REPORT
HMMMMM... NEW PROJECTIONS. I'LL CHECK THAT OUT AFTER I FINISH EMAIL.
SWAP
TEAM UP
#2 SAVED FILES
RIGHT NOW... BACK TO WORK!
CAROL JUST UPLOADED A NEW REPORT — A REVISION. OTTO NOTICES, AND TAKES A QUICK PEEK TO SEE WHAT CHANGED.
A PREVIEW POPS UP, AND IT LOOKS LIKE CAROL MADE AN IMPORTANT CHANGE THAT MIGHT BE WORTH LOOKING AT — LATER.
BUT FOR NOW, OTTO NEEDS TO FINISH WHAT HE'S DOING. TEAM UP DOWNLOADS THE FULL FILE AND HOLDS ON TO IT FOR HIM TO CHECK OUT LATER.

FIGURE 1-3. A storyboard (courtesy of Brandon Schauer)

User interface and visual design

Because the web is a visual medium, web pages require attention to their visual presentation. First impressions are everything. The "look and feel" of a site ensures that it makes a good first impression and is consistent with the brand and message of the organization it represents. These days, the visual design is usually handled by a **user interface** (**UI**) **designer** or **visual designer**.

UI design focuses on crafting the visual experience that website users interact with. In addition to designing the layout of components on pages at various sizes, designers apply the organization's brand identity (colors, logos, typography, and other styles) to those components in order to create websites that are impactful, effective, and accessible.

The creative team may also have a dedicated **graphic designer** who produces image assets and other collateral, or an **information designer** who specializes in visual assets meant to convey specific information, such as infographics, charts, and even interactive presentations.

There are many methods and deliverables that can be used to present a visual design to clients and stakeholders. The most traditional are sketches, often called **comps** or **mockups**, of the way the site might look, such as the home page comps shown in FIGURE 1-4. We'll talk about popular tools for designing site interfaces later in this chapter.

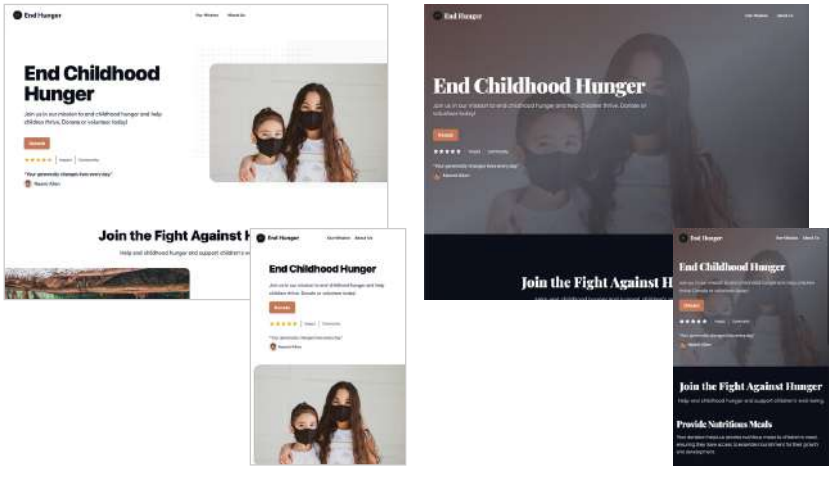


FIGURE 1-4. Look-and-feel comps for a simple site (designs generated on *Mixo.io*)

Now that sites appear on screens of all sizes, the standard practice is to take a component-driven approach to design. Page components such as page headers, search fields, navigation, calls to action, product galleries, and so on need to adapt and rearrange based on the available screen space, so UI designers

must think about how every page component performs in different contexts. It's about building a visual language for the site that can be applied to various page types at various sizes.

To be sure the components are used consistently by the whole team, they are commonly stored and documented in a design system. A **design system** is a collection of reusable site components, such as UI widgets, content containers, brand styles, and more, that can be shared by a team (I'll talk more about design systems in **Chapter 19, Responsive Web Design**). Each component generally includes a name, image, description, and code that can be copied and pasted. Creating and managing a design system and its UI component library has become central to the responsibilities of the modern UI designer.

Some designers prefer to get sign-off on the visual identity (colors, fonts, image style, etc.) in a way that isn't tied to a specific layout, like the comps shown in FIGURE 1-4. The designer may choose to create a deliverable similar to a mood board that gives an overview of the visual design at a glance, such as the **style tiles** approach introduced by Samantha Warren (FIGURE 1-5). Learn more about it at ***styletil.es***.

Designers may also need to produce the image assets for the site, so knowledge of how to optimize images for fastest delivery is a key skill. For large sites with lots of images, it is more common for the development team or a third-party service to handle image optimization, but this is a skill every visual designer should have. We'll discuss image optimization in **Chapter 27, Image Production Techniques**.

FIGURE 1-5. The style tiles technique, introduced by Samantha Warren

What Employers Look for in a UX/UI Designer

User experience design is in high demand these days, as businesses acknowledge the effect that thoughtful, user-centered design has on the success of their products and services. Although it is possible to get a degree in UX design from a university, UX designers come from a wide variety of backgrounds. Many start their careers as visual or product designers; some have backgrounds in technology or human–computer interaction (HCI); others come at it with experience in psychology or marketing.

If you are interested in getting into UX design, you might consider attending a UX bootcamp, which is an accelerated program that gets your skills up to speed quickly. There are many of them out there, so shop around and compare.

Bachelor's degrees in areas such as design, computer science, information architecture, HCI, or psychology are good starting points on the path to UX design. In addition to a bachelor's degree, employers typically look for the following:

- Experience working on web and mobile products, preferably in a design role (or product management for UX design)
- Familiarity with industry design tools such as Figma and the Adobe Suite
- User research and testing experience
- Experience with UX deliverables such as site diagrams, personas, flowcharts, etc.
- An understanding of business metrics (especially for UX design)
- A working knowledge of HTML, CSS, and JavaScript
- Familiarity with site-building tools like Bootstrap, Tailwind CSS, or WordPress
- Strong multitasking and organizational skills, and attention to detail
- Clear communication skills

Writing the Code

A large share of the website building process involves creating and troubleshooting the documents, style sheets, scripts, databases, and images that make up a site. The team that handles the creation of the files that comprise the site (or templates for pages that get assembled dynamically) is usually called the **development** or **production department**.

Development falls under two broad categories: frontend development and backend development (but see the sidebar **"The Frontend and Backend Lines Have Blurred"**). Once again, these tasks may fall to specialists, but it is just as common for one person or team to handle both responsibilities.

The Frontend and Backend Lines Have Blurred

In truth, the divide between frontend and backend development isn't as clean as it once was. For example, JavaScript can now be used for just about everything, on both the frontend and the backend. For the purposes of this discussion, I've introduced JavaScript in the frontend category based on its original intent, but know that JavaScript skills are central to what has traditionally been considered backend programming as well. The lines have blurred.

Brad Frost gives his take on how development roles have evolved in his article "Front-of-the-Front-End and Back-of-the-Front-End Web Development" (*oreil.ly/laaJt*).

Frontend development

Frontend developers are responsible for all code that appears in or relates directly to the web browser. That includes HTML, CSS, JavaScript, and related production frameworks. You will need to have *intricate* knowledge of these technologies if you want a job as a web developer. Let's take a quick look at each.

Authoring/markup (HTML)

HyperText Markup Language (**HTML**) is the authoring language used to create web page documents. **Authoring** is the process of preparing content for delivery on the web, or more specifically, marking up the content with HTML tags that describe it.

HTML is not a programming language; it is a **markup language**, which means it's a system for identifying and describing the various components of a document, such as headings, paragraphs, and lists. The markup indicates the document's underlying **structure** (you can think of it as a detailed, machine-readable outline). You don't need programming skills to write HTML, just patience and common sense.

The best way to learn HTML is to type out some pages by hand, as we will be doing in the exercises in **Part II, HTML for Structure**.

Styling (CSS)

While HTML is used to describe the content of a web page, **Cascading Style Sheets** (**CSS**; see **Note**) describes how that content should be displayed. The way the page looks is referred to as its **presentation**. Fonts, colors, background images, line spacing, page layout, margins, and so on are all controlled with CSS. You can even use it to add special effects and basic animation to your pages.

The CSS specification also provides methods for controlling how documents will be presented in contexts other than a browser, such as in print or when read aloud by a screen reader; however, I won't be covering them much here.

NOTE

When this book uses the term "style sheets," it always refers to Cascading Style Sheets, the standard style sheet language for the World Wide Web. Style sheets (including what "cascading" means!) are discussed further in **Part III, CSS for Presentation**.

The World Wide Web Consortium

The **World Wide Web Consortium** (called the **W3C** for short) is the organization that oversees the development of web technologies such as HTML, CSS, XML, and many more, as well as standards for web accessibility, internationalization, privacy, and security. The group was founded in 1994 by Tim Berners-Lee, the inventor of the web, at the Massachusetts Institute of Technology (MIT).

In the beginning, the W3C concerned itself mainly with the HyperText Transfer Protocol (HTTP) and the development of HTML. Now, it's laying a foundation for the future of the web by developing dozens of technologies and protocols that must work together in a solid infrastructure. For the definitive answer to any web technology question, the W3C site (*w3.org*) is the place to go. For more information on the W3C and what it does, see *w3.org/about/*.

The HTTP protocol is now managed by the Internet Engineering Task Force (IETF) HTTP Working Group (*httpwg.org*). In 2019, the W3C handed over development and maintenance of HTML to the **Web Hypertext Application Technology Working Group** (**WHATWG**; *whatwg.org*), which was founded by representatives of leading browser vendors in 2004, but the spec is published on the W3C site as well.

Although it is possible to publish web pages using HTML alone, you'll probably want to get comfortable using style sheets so you're not stuck with the browser's default styles. If your goal is designing websites professionally, either as a designer or as a developer, proficiency with CSS is mandatory.

JavaScript and DOM scripting

JavaScript is a scripting language that was originally developed to add interactivity and behaviors to web pages, which is why I'm introducing it in the context of frontend development. However, over the years it has also become increasingly popular and important for backend programming. And not only that: JavaScript has had a huge impact beyond the web as well! It powers many popular applications, such as Figma, Photoshop, Netflix, social media platforms like Facebook, and many more.

In the context of frontend web development, here are just a few interactions handled by JavaScript:

- Checking form entries for valid entries
- Swapping out styles for an element or an entire site
- Loading scrolling feeds with more content automatically
- Making the browser remember information about users
- Building interface widgets, such as embedded video players or special form inputs
- Adding animation effects

■ AT A GLANCE

Frontend Development

Frontend development includes the following web technologies:

- HTML
- CSS
- JavaScript and DOM scripting, including JavaScript frameworks

You may also hear the term **DOM scripting** used in relation to JavaScript. DOM stands for **Document Object Model**, and it refers to the standardized list of web page elements that can be accessed and manipulated using JavaScript (or another scripting language).

Frontend developers should also be familiar with JavaScript frameworks (such as React, Vue, Angular, Node.js, and others) that provide helpful structure and conventions that make writing JavaScript easier. They simplify and streamline the production process.

Web scripting definitely requires some traditional computer programming prowess. While many web developers have degrees in computer science, it is also common for developers to be self-taught. A few developers I know started by copying and adapting existing scripts, then gradually added to their scripting skills with each new project. Still, if you have no experience with programming languages, the initial learning curve may be a bit steep.

If you want to be a web developer for a living, JavaScript is a basic requirement. Designers will benefit from understanding what JavaScript can do, but may not need to learn to write it if they are working with a development team. The chapters in **Part IV, JavaScript for Behavior**, will get you started.

Backend development

Backend developers focus primarily on the server, including the applications and databases that run on it. They may be responsible for installing and configuring the server software (we'll be looking more at servers in **Chapter 2, How the Web Works**). They will certainly be required to know one or more server-side programming languages, such as PHP, Ruby, Node.js, .NET (or ASP.NET), and Python, in order to create applications that provide

What Employers Look for in a Web Developer

As you may expect, employers want to know that prospective web developers are well trained in web-related technologies. You can get that training as part of a traditional computer science college degree, through a web development bootcamp, or even through prior experience in the workplace. There are also web development certification programs available out there (for a fee, of course); although certification certainly doesn't hurt, it is not often a requirement in job listings.

Specific software and programming language requirements vary by company (for example, some are all-in on Microsoft products), but here is a list of requirements that typically appear in web developer job posts:

- Deep understanding of HTML and CSS
- Experience with programming languages such as JavaScript, Java, C++, or Python
- Experience with modern web development platforms such as React or Angular
- Familiarity with version control management tools, such as Git
- Microsoft .NET experience or certification (depending on context)
- Database programming experience (depending on context)
- Problem solving and analytical skills
- Clear communication skills and ability to work in a team

When looking for a job in web development, you will frequently see posts looking for **"full-stack" developers**, meaning people who are fluent in both frontend (HTML, CSS, JavaScript) and backend (server software, databases) technologies, as well as possibly having additional UX design skills.

the functionality required by the site. It is worth noting that JavaScript is used for applications that run on the server side as well. Server-side applications handle tasks and features like forms processing, content management systems (CMSs), and online shopping, just to name a few.

Additionally, backend developers need to be familiar with configuring and maintaining databases that store all of the data for a site, such as the content that gets poured into templates, user accounts, product inventories, and more. Some common database languages include MySQL, Postgres, NoSQL, Oracle, and SQL Server.

Backend development is well beyond the scope of this book, but it's important to know the sorts of tasks that get taken care of at the server level. You should be aware that it is possible to get functionality like shopping carts, mailing lists, and so on as prepackaged solutions from your hosting company without needing to program it from scratch.

AT A GLANCE

Backend Development

The following technologies are typically in the domain of the backend developer:

- Server software (Apache, Microsoft IIS, nginx, and more)
- Web application languages (Node.js, PHP, Ruby, Python, JSP, ASP.NET, JavaScript)
- Database software (MySQL, Oracle, SQL Server, Postgres, NoSQL)

Content Wrangling

Anyone who uses the title "web designer" or "web developer" needs to be aware that everything we do supports the process of getting our content, message, or functionality to our users. Furthermore, good writing can help the user interfaces we create be more effective, from button labels to error messages.

Of course, someone needs to create all that content and maintain it—don't underestimate the resources required to do this successfully. Good writers and editors are an important part of the site creation team. In addition, I want to call your attention to two content-related specialists in modern web development: the information architect (IA) and the content strategist.

Information architecture

An **information architect** organizes the content logically and for ease of findability. They may be responsible for search functionality, site diagrams (such as the one shown in FIGURE 1-2), and how content and data are organized on the server. Information architecture is inevitably entwined with UX and UI design as well as content management. If you like organizing or are enthusiastic about taxonomies, information architecture may be the job for you. For more information on this field as it relates to the web, the book *Information Architecture: For the Web and Beyond*, 4th edition, by Louis Rosenfeld, Peter Morville, and Jorge Arango (O'Reilly) will serve you well.

Content strategy

When the content isn't right, a site can't be fully effective. A **content strategist** makes sure that every bit of text on a site, from long explanatory text down to the labels on buttons, supports the brand identity and marketing goals of the

What Employers Look for in a Web Content Strategist

If guiding a company's content to make sure it's consistent with its brand identity and business goals sounds like an exciting way to make a living, you may want to become a content strategist. Although job requirements vary by company and position, employers generally look for some combination of the following qualifications:

- A bachelor's degree in subjects like (but not limited to) English, technical writing, marketing, communications, or user experience
- Experience in marketing and branding
- Experience with project or account management
- Strong organizational skills and attention to detail
- Communication skills, both speaking and writing
- Problem solving and strategic thinking
- Leadership skills and the ability to work well as part of a creative team

organization. Content strategy may also extend to data modeling and content management on a large and ongoing scale, such as planning for content reuse and updating schedules. The content strategist's responsibilities might also include how the organization's voice is represented on social media. For more reading about content strategy, try *From Solo to Scaled: Building a Sustainable Content Strategy Practice* by Natalie Marie Dunbar (Rosenfeld Media), or search for books written by Kristina Halvorson and Margot Bloomstein.

Other Roles

Not surprisingly, there are a myriad of other roles that contribute to the creation and maintenance of a site, all of which benefit from some understanding of the basics of web development as taught in this book. These include:

Product manager

The product manager of a website or application guides its design and development in a way that meets business goals. This member of the team must have a thorough understanding of the target market as well as the processes involved in the creation of the site itself. Product managers develop the overall strategy for the site from a marketing perspective, including how and when it gets released.

Project manager

The project manager coordinates the designers, developers, and everyone else who is working on the site. They manage things like timelines, development approaches, deliverables, and so on. The project manager works with the product manager and other product owners to make sure that the project gets done on time and on budget.

SEO specialist

A website or application isn't much good if nobody knows it exists, so it is crucial that a site be easily found by search engines. **Search engine optimization** (**SEO**) is a discipline focused on tweaking the site structure and code in a way that increases the chances that it will be highly ranked in search results. There may be an SEO specialist on the in-house team, or a company may choose to hire an outside SEO firm. SEO is sometimes perceived as a dark art, but there are many ways to improve findability that are not underhanded. In fact, the number one technique for improving SEO is simply having good content with savvy HTML markup.

Multimedia producers

One of the cool things about the web is that you can add multimedia elements to a site, including sound, video, animation, and even interactive games. Creating multimedia elements is generally best left to artists and technicians in those fields, although they may be part of the web team if video, animation, or interactivity is core to the site's mission.

That concludes our stroll through the virtual village of workers involved in the creation of a website. The larger the site, the more likely it is that each team member will have a narrow specialization and a job title like "UX Lead for Error Messages." More likely, everybody on the team will possess a spectrum of skills, and the lines between disciplines will blur. For example, I do interaction and user interface design, visual design, HTML, and CSS, but I don't write JavaScript or work on the server.

TOOLS OF THE TRADE

It should come as no surprise that professional web designers require a fair amount of gear, both hardware and software. One question frequently asked is, "What do I need to buy?" I can't tell you specifically, but I will provide an overview of the typical tools of the trade.

Equipment

For a comfortable web development environment, I recommend the following equipment:

A solid, up-to-date computer

Use whatever you have and are comfortable with—macOS, Windows, or Linux is fine. Design departments tend to be Mac-based. For development

(Soft) Skills Every Web Designer Needs

We've focused on quite a few technical skills that will be helpful in building websites. I'd like to mention a few more (often overlooked) skills that are just as critical to your success:

Communication skills

In your work, you will need to communicate in person, on the phone, in email, and in text messaging tools with clients, team members, and superiors. Be clear, proactive, and straightforward with what you have to say. Good communication requires not only that you express yourself clearly, but also that you be a good listener. Make sure that you understand the issues being discussed, and don't be afraid to ask for clarification if you don't.

Collaboration

Remaining open-minded to new ideas and willing to compromise for the greater good are keys to successful collaboration.

Flexibility

It's important to be able to change direction quickly, because not only do web technologies evolve rapidly, but you will no doubt be thrown curveballs in your day-to-day work as well. For example, you may arrive at work one day to find that the client has changed your priorities completely. You might even find that they've cancelled your project entirely. You may be asked to learn new skills and shift positions in the team. Staying adaptable is the key to survival.

Critical thinking and good judgment

Problem solving is central to all of the disciplines related to web design, so you need to be able to use critical thinking skills to come up with solutions and always employ basic common sense.

A good attitude

Creating sites means being part of a team, even if you work at home as a freelancer. Be mindful that the attitude with which you approach your work is contagious, so strive to be a positive and friendly team member.

work, Windows and Linux are popular (although many developers choose to use Macs as well). Although it's nice to have a super-fast machine, the files that make up web pages are small and tend not to be too taxing on computers. Unless you're getting into sound and video editing, don't worry if your current setup is not the very latest and greatest.

A large monitor

Though it's not a requirement, a large monitor makes life easier. The more monitor real estate you have, the more windows and control panels you can have open at the same time. If you're using a large monitor, just make sure you design for users with smaller screens in mind.

A second computer for testing

Many designers and developers find it useful to have a test computer running a different platform than the computer they use for development (i.e., if you design on a Mac, test on a PC). Because sites may work differently on Macs than on Windows machines, it's critical to test your pages in as many environments as possible. If you are a hobbyist web designer working at home, you can install an alternate operating system on your Mac or Windows PC, as discussed in the **"Run Windows on Your Mac (and Vice Versa)"** sidebar. Or go visit a friend and check your pages on their computer.

Mobile devices for testing

It is absolutely critical that you test the appearance and performance of your site on browsers on smartphones and tablet devices. Device testing is discussed further in **Chapter 19**.

Run Windows on Your Mac (and Vice Versa)

You may not need to use a separate computer to have access to both Windows and macOS. There are ways to install the alternate operating system on your machine with virtual machine (VM) software.

On newer Macs with M1 or M2 chips, you can use a software program called Parallels (*parallels.com*) to run Windows 11 (the latest version as of this writing) in a separate partition, so you can switch between operating systems. Macs with Intel chips can run Boot Camp to install Windows, but they can only boot up in one OS or the other, not switch back and forth. All VM products for Mac require that you purchase a copy of Microsoft Windows, but it sure beats buying a whole machine.

There are also several methods for installing macOS on computers running Windows 10 or 11. One popular method is to download and install Oracle VirtualBox (*virtualbox.org*) and create a partition to run macOS. The full instructions are longer than I can address here, but a quick web search will turn up several helpful tutorials.

Web Production Software

There's no shortage of software available for creating web pages. In the early days, we just made do with tools originally designed for print. Today, there are wonderful tools created specifically with web design in mind that make the process more efficient. It's a delicate business listing software in a book such as this, because a) there are *so many* programs, b) everyone has their personal favorite, and c) new tools come along so rapidly that as you're reading this chapter there are likely already newer, cooler options available that didn't exist as I wrote it.

That said, here is a general overview of the types of software that comprise the tools of our trade, along with a few specific mentions of the most popular in each class (at the time of writing).

Coding tools

Although you can get by with the simple text editors that come with your computer, a dedicated code editor makes the task of writing HTML, CSS, and

JavaScript much easier. Code editors understand the syntax of the code you write, so they can do things for you like color coding, error detection, code validation, and automatically finishing simple tasks like closing HTML tags. Some provide page previews so you can view the results of your code as you work.

Here are just a few of the better-known code editors for web production that are worth exploring:

- Visual Studio Code (***code.visualstudio.com***; free)
- Phoenix Code (***phcode.dev***; a free browser-based code editor with a page preview)
- Sublime Text (***sublimetext.com***)
- Adobe Dreamweaver (***adobe.com/products/dreamweaver.html***)
- CodeKit (***codekitapp.com***; *Mac only*)

FIGURE 1-6 shows how an HTML document with its embedded style sheet looks in the free Visual Studio Code editor.

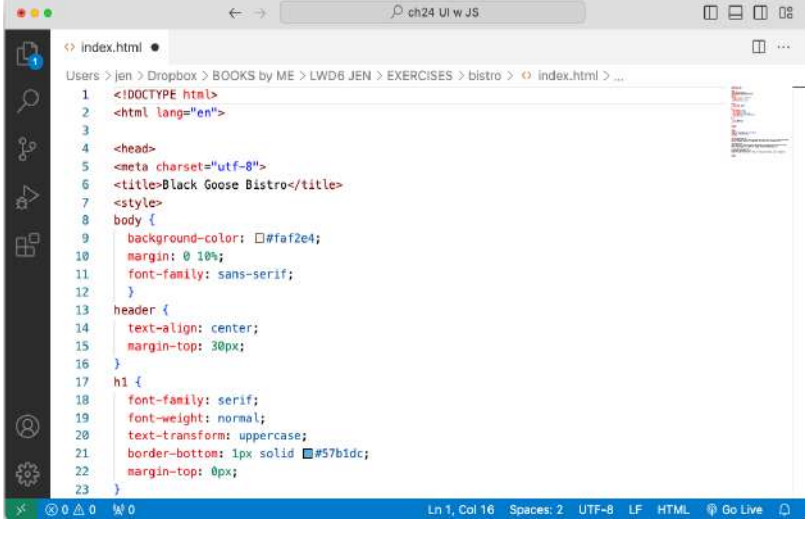


FIGURE 1-6. The Visual Studio Code editor

Another great tool is CodePen (***codepen.io***), an easy-to-use online code editor that shows a preview of your page as you work. CodePen is much more than just a code editor, though; it's also a community where people share their work and make it available to others. For beginners, it's a great place to learn from code examples. It's free to get started, but if you want to host images and other files and get access to advanced features, there is a monthly or annual fee.

NOTE

To do the exercises in this book, all you'll need is the text editor that came with your operating system and free image creation software. There is no need to purchase anything to follow along.

User interface and layout tools

There is a new breed of user interface design tools made specifically for websites and other applications. Because they have been designed from the ground up with interface design in mind, they seem to anticipate a web designer's every need. Interface design tools make it easy to design multiple layouts (such as layouts at various screen sizes) and to export images and code for use in production. Some allow basic interactivity, such as clicks and swipes, so your prototypes can be shared online and used for basic interface testing.

Figma (***figma.com***), shown in FIGURE 1-7, is by far the most popular design tool at the time of writing because of its prototyping capabilities, developer mode, cross-platform support, and ease of collaboration. Other options include the following:

- Sketch (***sketch.com***; *Mac only*)
- UXPin (***uxpin.com***)

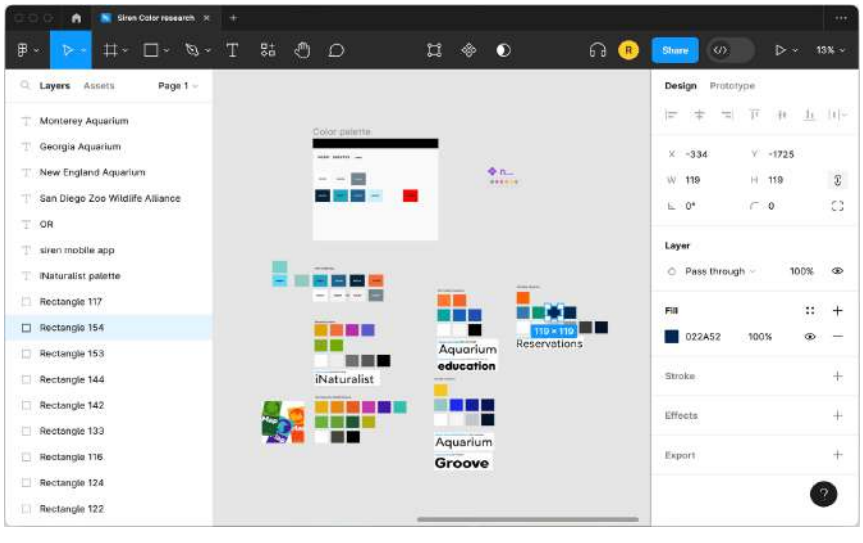


FIGURE 1-7. The Figma user interface design tool

Image editing tools

It's certainly possible to create all the images you need for a site by using one of the interface design tools just listed. There are also programs that focus solely on image creation that can export files in web-appropriate formats. For professional designers, the Adobe Creative Cloud (***adobe.com***) suite of tools, which includes Photoshop (FIGURE 1-8), Illustrator, and other high-end design tools, may be worth the investment.

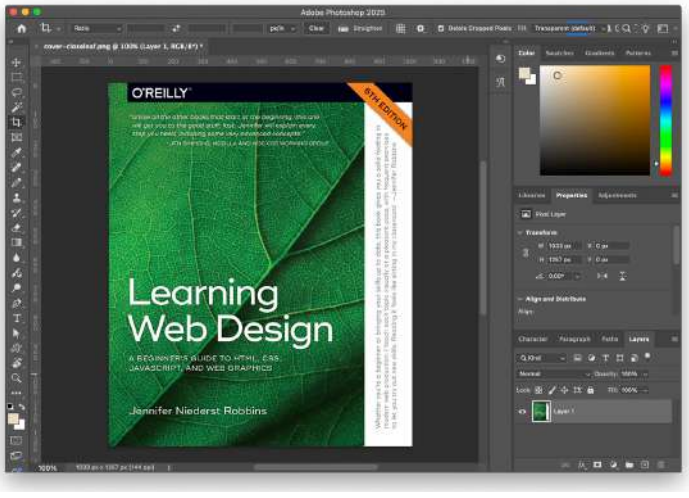


FIGURE 1-8. Adobe Photoshop: the professional standard for image editing

If the Adobe monthly subscription fee is out of reach, you can try lower-cost alternatives that provide many of the same features. The number of graphics tools out there is dizzying, so I'll mention just a few here:

- GIMP (free, open source; ***gimp.org***; *all platforms*)
- Corel PaintShop Pro (for photo editing; ***paintshoppro.com***; *Windows only*)
- Corel Draw (for vector drawing; ***coreldraw.com***; *Windows only*)
- Pixelmator Pro (***pixelmator.com***; *Mac only*)

The following image editors work right in your browser, without the need to download a program:

- SumoPaint (***sumopaint.com***; free online tool; paid Pro version also available)
- Pixlr (***pixlr.com***; free)
- Photopea (***photopea.com***; free)

A variety of browsers

One of the biggest challenges for web designers is that our sites may look and behave differently from browser to browser. For this reason, it is critical that we test our designs early and often on the widest range of browsers possible, both desktop and mobile. These are the most popular browsers, and the ones designers and developers typically keep around for testing:

Desktop

- Chrome (***google.com/chrome***)
- Safari (***apple.com/safari***; *Mac only*)

- Microsoft Edge (*microsoft.com/edge*; see **Note**)
- Firefox (*mozilla.org*)
- Opera (*opera.com*)

Mobile

- Chrome
- Safari (Apple App Store only)
- Samsung Internet (Google Play only)
- Opera Mini
- Firefox
- Microsoft Edge

Unless otherwise noted, these mobile browsers are available for download on the Apple App Store and Google Play. I'll discuss mobile testing further in **Chapter 19**.

NOTE

Although once dominating browser statistics with over 95% of web traffic, Internet Explorer usage has reduced to less than 1% now that Microsoft has discontinued it in favor of Edge. There may be special circumstances under which you need to support Internet Explorer, and if that is the case, you will find many articles online describing what it does and doesn't support, as well as workarounds. In general, I will not be providing Internet Explorer–related information in this book, with a few exceptions.

File management and transfer tools

Web design and development involves a lot of moving files around, particularly from the computer where you do your work to the server computer that hosts the site. There are various tools that come in handy for version control and file transfer.

Git and GitHub

These days, most web development teams use a version control management program called Git. Git keeps copies of every version of a document in a "repository" and allows you to roll back to previous iterations. It also makes it very easy for teams to share and contribute to a central codebase. Many organizations and individuals use the online service GitHub to host their Git repositories (you'll see *github.com* in the links for many resources in this book). In the modern web development process, files are most often deployed to the server via Git, and it has become central to the development process.

Knowledge of Git is critical if you are on a professional team and useful even if you are working alone. I recommend taking the time to read my brief introductory article "Version Control with Git and GitHub," available at *learningwebdesign.com/articles*.

FTP

The traditional way to move files across the internet is to use an **FTP** (short for **File Transfer Protocol**) program. If you are not working on a team using Git or more elaborate build and deployment tools, you can use a dedicated FTP application to move your files to the server. You will find that many hosting services offer their own FTP tools for uploading your files. Many of

the code editors listed earlier also include built-in FTP functionality. A few popular standalone FTP programs include:

- Filezilla (***filezilla-project.org***; free; *all platforms*)
- Cyberduck (***cyberduck.io***; *Mac and Windows*)
- WinSCP (***winscp.net***; free; *Windows only*)
- Transmit (***panic.com/transmit/***; *Mac only*)

Command-line tools

You may also find it useful to have a **terminal application** (command-line tool) that allows you to type Unix commands for setting file permissions, moving or copying files and directories, or managing the server software. Command-line tools have a number of uses in web design and development workflows. For a quick introduction, see my article "An Introduction to the Command Line," available at ***learningwebdesign.com/articles***. The following are examples of terminal applications:

- Terminal (installed with macOS; shown in FIGURE 1-9)
- Cygwin (***cygwin.com***; Linux emulator for Windows that includes a command-line tool)

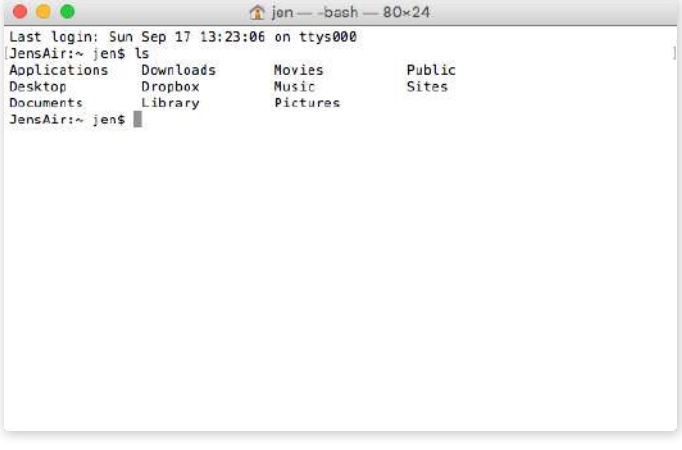

jen — -bash — 80×24
Last login: Sun Sep 17 13:23:06 on ttys000
JensAir:~ jen$ ls
Applications Downloads Movies Public
Desktop Dropbox Music Sites
Documents Library Pictures
JensAir:~ jen$

FIGURE 1-9. The Terminal command-line tool for macOS

WHAT YOU'VE LEARNED

This chapter has given you an overview of the many roles and responsibilities that contribute to the creation of websites. Hopefully, you've come away realizing that you don't need to learn everything—and even if you do want to learn everything eventually, you don't need to learn it all at once. So relax,

EXERCISE 1-1.
Taking stock

Now that you're taking that first step in learning web design, it might be a good time to take stock of your assets and goals. Using the lists in this chapter as a general guide, try jotting down answers to the following questions:

- What are your web design goals? To become a professional web designer? To make personal websites only? To work in a management position?
- Which aspects of web design interest you the most? Writing code? Creating the look and feel and visual assets? Organization? Everything?!
- What current skills do you have that will be useful in creating web pages?
- Which skills will you need to brush up on?
- Which hardware and software tools do you already have for web design?
- Which tools do you need to buy? Which tools would you like to buy eventually?

and don't worry. The other good news is that, while many professional tools exist, it is possible to create a basic website and get it up and running without spending any money by using freely available tools and your existing computer setup.

As you'll soon see, it's easy to get started making web pages—you will be able to create simple pages by the time you're done reading this book. From there, you can continue adding to your bag of tricks and find your particular niche in the website creation ecosystem. In the meantime, try answering the questions in EXERCISE 1-1.

TEST YOURSELF

Each chapter in this book ends with a few questions that you can answer to see if you picked up the important bits of information. Answers appear in **Appendix A**.

1. Match these web professionals with the final products they might be responsible for producing:

a. UI/visual designer	____ HTML and CSS documents
b. Production department	____ Setting up a SQL database
c. User experience designer	____ "Look and feel" deliverables
d. Backend programmer	____ User research reports

2. What does the W3C do?

3. Match each web technology with its respective task:

a. HTML	____ Checks a form field for a valid entry
b. CSS	____ Creates a custom server-side web application
c. JavaScript	____ Identifies text as a second-level heading
d. PHP	____ Makes all second-level headings blue

4. What is the traditional difference between frontend and backend web development?

5. What tool do many web development teams use for version control?

6. Do designers need to know how to code?

CHAPTER
2

HOW THE WEB WORKS

I got started in in web design in early 1993—pretty close to the start of the web itself. That's a roughly a third of a century ago (gasp!), but I still distinctly remember the first time I looked at a web page. It was difficult to tell where the information was coming from and how it all worked.

This chapter sorts out the pieces and introduces some important terminology. We'll start with the big picture and work down to specifics.

Here is what you can expect to learn in this chapter:

- The difference between the internet and the web
- The role of the server and browser
- The structure of URLs (web page addresses)
- How HTML and style sheets work together
- How the browser communicates with the server when you go to a web page

IN THIS CHAPTER

An explanation of the web as it relates to the internet

The role of the server

The role of the browser

URLs and their components

The anatomy of a web page

THE INTERNET VERSUS THE WEB

No, it's not a battle to the death, just an opportunity to point out the distinction between two words that are often used interchangeably.

The **internet** is an international network of connected computers. No company owns the internet; it is a cooperative effort governed by a system of standards and rules. The purpose of connecting computers together, of course, is to share information. There are many ways information can be passed between computers, including via email (POP3/IMAP/SMTP), file transfer (FTP), Secure Shell (SSH), and various other specialized modes upon

The web is a subset of the internet. It is just one of many ways information can be transferred over networked computers.

which the internet is built. These standardized methods for transferring data or documents over a network are known as **protocols**.

The **web** (originally called the World Wide Web, thus the "www" in site addresses) is just one of the ways information can be shared over the internet. It is unique in that it allows documents to be linked to one another via **hypertext** links, thus forming a huge "web" of connected information. The web uses a protocol called **HTTP** (**HyperText Transfer Protocol**). That acronym should look familiar because it is the first four letters of nearly all website addresses, as we'll discuss in an upcoming section.

SERVING UP YOUR INFORMATION

Let's talk more about the computers that make up the internet. Because they "serve up" documents upon request, these computers are known as **servers**. More accurately, the server is the software (not the computer itself) that allows the computer to communicate with other computers; however, it is common to use the word "server" to refer to the computer as well. The role of server software is to wait for a request for information, and then retrieve and send that information back as quickly as possible.

In order for a computer to be part of the web, it must be running special web server software that allows it to handle HTTP transactions. Web servers are also called **HTTP servers**.

There are many server software options out there, but the two most popular are Apache and nginx (both **open source** software; see the sidebar). Both Apache and nginx are freely available for Unix-based computers and Windows, though they're most often run on Unix-based machines. Apache also comes installed by default on Macs running macOS.

A Brief History of the Web

The web was born in a particle physics laboratory (CERN) in Geneva, Switzerland, in 1989. There, a computer specialist named Tim Berners-Lee first proposed a system of information management that used a "hypertext" process to link related documents over a network. He and his partner, Robert Cailliau, created a prototype and released it for review. For the first several years, web pages were text-only. By 1992, the world had only about 50 web servers, total.

The real boost to the web's popularity came in 1992 when the first graphical browser (NCSA Mosaic) was introduced, and the web broke out of the realm of scientific research into commercial mass media. The ongoing development of web technologies is overseen by the World Wide Web Consortium (W3C).

If you want to dig deeper into the web's history, check out the W3C's history archives at *w3.org/History.html*.

Open Source

Open source software is developed as a collaborative effort with the intent to make its source code available to other programmers for use and modification. By allowing anyone to access, modify, and distribute the source code, open source fosters collaboration and a sense of community among developers worldwide. Because developers can build on each other's work rather than starting from scratch, it accelerates the pace of development. It also leads to improved software quality, as many eyes can identify bugs.

Open source programs are usually free to use, so startups and individuals can create and use powerful tools without the high cost of proprietary solutions. Examples of open source software include the Linux operating system, the Bootstrap web development framework, the Mozilla Firefox browser, the Git version control management system, the Drupal content management system, and many more.

Anyone can contribute to an open source project. It's a great way to learn, discover, and be a part of the tech community.

Every computer and device (router, smartphone, car, etc.) connected to the internet is assigned a unique numeric **Internet Protocol (IP) address**. For example, as I write this, the computer that hosts ***oreilly.com*** has the IP address 199.27.145.64. All those numbers can be dizzying, so fortunately, the **Domain Name System** (**DNS**) was developed to allow us to refer to that server by its **domain name**, "oreilly.com," as well. The numeric IP address is useful for computer software, while the domain name is more accessible to humans. Matching the text domain names to their respective numeric IP addresses is the job of a separate **DNS server**. It is possible to configure a web server so that more than one domain name is mapped to a single IP address, allowing several sites to share a single server.

A WORD ABOUT BROWSERS

You now know that the server does the serving, but what about the other half of the equation? The software that does the requesting is called the **client**. People use desktop browsers, mobile browsers, and other assistive technologies (such as screen readers) as clients to access documents on the web. The server returns the documents for the browser (also referred to as the **user agent** in technical circles) to display.

The requests and responses are handled via the HTTP protocol, mentioned earlier. Although we've been talking about "documents," HTTP can be used to transfer images, movies, audio files, data, scripts, and all the other web resources that commonly make up websites and applications.

It's common to think of a browser as a window on a computer monitor with a web page displayed in it. These are known as **graphical browsers** or **desktop browsers**, and for a long time, they were the only web-viewing game in town. The most popular desktop browsers as of this writing include Chrome, Safari, Edge, and Firefox (Opera and old versions of Internet Explorer make up a tiny percentage of usage).

These days, however, more than half of web traffic comes from mobile browsers on smartphones and tablets such as Safari on iOS, Android and Chrome browsers on Android devices, Opera Mini, and a myriad of other default and installable mobile browsers (see ***en.wikipedia.org/wiki/Mobile_browser*** for a complete list). Navigating the web on a touchscreen is the new normal.

In this book and in the HTML/CSS specs, you will commonly see the more technical term **viewport** to refer to the area in which a web page is displayed. The viewport may be a resizable browser window on a computer monitor, or the entire screen of a handheld device displaying a browser application.

It's also important to keep alternative web experiences in mind. Users with impaired sight may be listening to a web page read by a screen reader (or simply make their text extremely large). Users with limited mobility may use assistive technology such as joysticks or voice commands to access links and

TERMINOLOGY

Intranets and Extranets

When you think of a website, you generally assume that it is accessible to anyone surfing the web. However, many organizations take advantage of the awesome information sharing and gathering power of websites to exchange information just within their own networks. These private web-based networks are called **intranets**. They are created and function like ordinary websites, but they use special security devices called **firewalls** that prevent the outside world from seeing them. Intranets have lots of uses, such as sharing human resource information or providing access to inventory databases.

An **extranet** is like an intranet, but it allows access to select users outside of the organization. For example, a manufacturing company may provide its customers with passwords that allow them to check the status of their orders in the company's orders database. Passwords determine which slice of the company's information is accessible.

TERMINOLOGY

Server Side and Client Side

Often in web design, you'll hear references to "client-side" or "server-side" applications. These terms are used to indicate which machine is doing the processing. **Client-side** applications run on the user's machine (also referred to as the **frontend**), while **server-side** applications and functions use the processing power of the server computer (the **backend**).

Browser Rendering Engines

The program that is responsible for converting HTML, CSS, and JavaScript into what you see rendered on the screen is called a **rendering engine** (a.k.a. **browser engine** or **layout engine**). Browsers are made up of rendering engines as well as other code used for their own user interfaces and functionality. Although I talk a lot about which browsers support particular functions in this book, I'm technically referring to the browser's rendering engine. Multiple browsers often share a rendering engine; for example, the Blink engine powers Chrome, Microsoft Edge, Opera, and a variety of Android browsers. TABLE 2-1 lists the rendering engines used by the most popular web browsers today. For more information, search *Wikipedia.org* for "comparison of web browser engines" and "comparison of web browsers."

TABLE 2-1. Current browsers and their rendering engines

Browser	Rendering engine
Chrome 28+	Blink (forked from WebKit)
Firefox (all)	Gecko (except Firefox for iOS, which uses WebKit)
Safari and Safari iOS (all)	WebKit
MS Edge (79+)	Blink (forked from WebKit)
Opera 15+	Blink (forked from WebKit)
Internet Explorer 4–11	Trident

Pages may look and perform slightly differently from browser to browser.

enter content. The sites we build must be accessible and usable for all users, regardless of their browsing experiences.

The web is also finding its way onto smart TVs and gaming systems, where users access our pages with TV remotes or Xbox controllers. You never know where the web will pop up next!

The reality is that pages may look and perform differently from browser to browser. This is due to varying support for web technologies, varying device capabilities, and the users' ability to set their own browsing preferences. It is one of the most challenging aspects of designing and developing for our medium (see the **"Support and Statistics"** sidebar for tools that help.

WEB PAGE ADDRESSES (URLS)

Every page and resource on the web has its own special address called a **URL**, which stands for **Uniform Resource Locator**. It's nearly impossible to get through a day without seeing a URL (pronounced "U-R-L," not "erl") plastered on the side of a bus, printed on a business card, or broadcast on a television commercial. Web addresses are fully integrated into the modern vernacular.

Some URLs are short and sweet. Others may look like crazy strings of characters separated by dots (periods) and slashes, but each part has a specific purpose. Let's pick one apart.

The Parts of a URL

A complete URL is generally made up of three components: the protocol, the site name, and the absolute path to the document or resource, as shown in FIGURE 2-1.

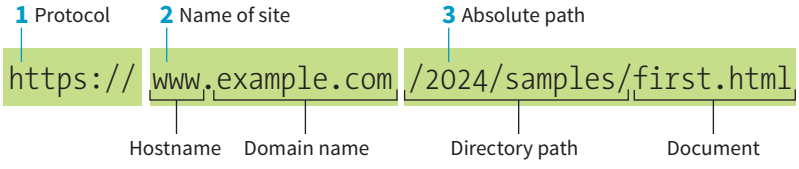


FIGURE 2-1. The parts of a URL

Let's break this example down:

1 `https://`

The first thing the URL does is define the protocol that will be used for that particular transaction. The letters "https" let the server know to use **HTTPS** (**HyperText Transfer Protocol Secure**), or get into secure "web mode" (my made-up term). This is one of many protocols used to transfer information over the internet. Defining the protocol as HTTPS indicates the client is requesting the transfer of encrypted text and data for a web page.

You may also see a URL begin with **`http://`** (no "s"), but HTTP is rapidly fading away in favor of the security provided by HTTPS. You can learn a little more in the **"Moving from HTTP to Secure by Default"** sidebar.

2 `www.example.com`

The next portion of the URL identifies the website by its **domain name**. In this example, the domain is "example.com." The "www." part at the

Support and Statistics

New HTML, CSS, and other web technology features are being added all the time, and it takes browsers a while to support them. And even when new browsers add support, it takes the older browsers some time to fade away. You'll see me mention browser support frequently in this book. One online resource that is critical for navigating the support landscape is Can I Use (***CanIUse.com***). You can search for the feature you want to use, and the site provides support information for all desktop and mobile browsers. There is also a view that shows support by year rather than browser version.

If you are curious about which browsers are most common and which are still hanging on, you can get up-to-date browser statistics at Statcounter (***gs.statcounter.com***). As the old web developer saying goes, however, "The only browser stats that matter are your own," so while the global stats provide a good starting point, paying attention to your own web traffic is the best way to make decisions about cutting-edge features.

Moving from HTTP to Secure by Default

Since the dawn of time, or at least the dawn of the web, the vast majority of traffic was served over the HTTP protocol. HTTPS was a rarity used only for particularly sensitive traffic, such as shopping online with a credit card or using a banking site.

They're very similar—in fact, HTTPS is an extension of HTTP—with the notable exception that HTTPS encrypts the data when it is sent between the user's client and the server, whereas in HTTP, data is sent as plain text.

As of this writing, around 80% of pages (and growing!) use HTTPS, and for good reason. Not only is it a good idea to keep your users' data secure in transit, but browser vendors have started to push much stronger incentives to use HTTPS. A lot of newer web technologies are available only if your site is served over HTTPS. Each major browser now also prominently displays some sort of "not secure" indicator in the address bar for any site using HTTP. In addition, Google looks for HTTPS when factoring in how to rank the site in its search engine.

HTTPS works in tandem with another protocol, **SSL** (for **Secure Sockets Layer**), which needs to be enabled on the server for secure transactions to work. While there used to be an added cost for enabling SSL, today most hosting companies have options for enabling it for free, and the Let's Encrypt project (***letsencrypt.org***) provides free SSL certificates for anyone.

Keep in mind that while HTTPS protects data as it is sent to the server, that doesn't mean your site is "secure" and safe from hackers. It's a start, but not a complete solution.

beginning is the particular **hostname** at that domain. The hostname "www" has become a convention, but it's not a rule. In fact, sometimes the hostname may be omitted. There can be more than one website at a domain (called **subdomains**). For example, there might also be sites at development.example.com, clients.example.com, and so on.

3 `/2024/samples/first.html`

This is the absolute path through directories on the server to the requested HTML document, *first.html*. The words separated by slashes are the directory names, starting with the root directory of the host (as indicated by the initial `/`). Because the internet originally comprised computers running the Unix operating system, our current way of doing things still follows Unix rules and conventions, hence the `/` separating directory names.

To sum it up, the URL in FIGURE 2-1 says it would like to use the HTTPS protocol to connect to a web server on the internet called "www.example.com" and to request the document *first.html*, located in the *samples* directory, which is in the *2024* directory.

Simplified URLs

Obviously, not every URL you see is so lengthy. To get to O'Reilly's site, for example, you'd expect to type `oreilly.com` instead of `https://www.oreilly.com/index.html`. Here's why that works.

Skipping the protocol

Because nearly all web pages use HTTPS, the `https://` part is often just implied. This is the case when site names are advertised in print or on TV, as a way to keep the URLs easy to remember.

Additionally, browsers are programmed to add `https://` automatically as a convenience to save you some keystrokes. It may seem like you're leaving it out, but it is being sent to the server behind the scenes.

When we begin using URLs to create hyperlinks in HTML documents in **Chapter 6, Adding Links**, you'll learn that it is necessary to include the protocol when making a link to a web page on another server.

Pointing to default files

Many addresses do not include a filename, but simply point to a directory, like these:

```
https://www.oreilly.com
https://www.example.com/resume/
```

When a server receives a request for a directory name rather than a specific file, it looks in that directory for a default document, typically named

index.html. So when someone types the previous URLs into their browser, these are the files they will actually see:

```
https://www.oreilly.com/index.html
https://www.example.com/resume/index.html
```

The name of the default file (also referred to as the **index file**) may vary, and depends on how the server is configured. In these examples, it is named *index.html*, but some server software uses the filename *default.htm*. If your site uses server-side programming to generate pages, the index file might be named *index.php* or *Default.aspx*. Check with your server administrator or the tech support department at your hosting service to make sure you give your default file the proper name.

Another thing to notice is that in the first example, the original URL did not have a trailing slash to indicate it was a directory. If the slash is omitted, the server checks to see if the request is a file or a directory. If it is a directory, the server asks the browser to send the request again with a slash. In the end, the slash is included for directories, even if it isn't included the first time it is entered.

MARKUP TIP

Include the trailing slash in the URLs in links to avoid a needless second trip to the server (a small hit to performance).

The index file is also useful for security. Some servers (depending on their configuration) display the contents of the directory if the default file is not found. FIGURE 2-2 shows how the documents in the *housepics* directory are exposed as the result of a missing default file. Your server administrator may add protections to prevent your directories from displaying in the browser if the index file is missing.

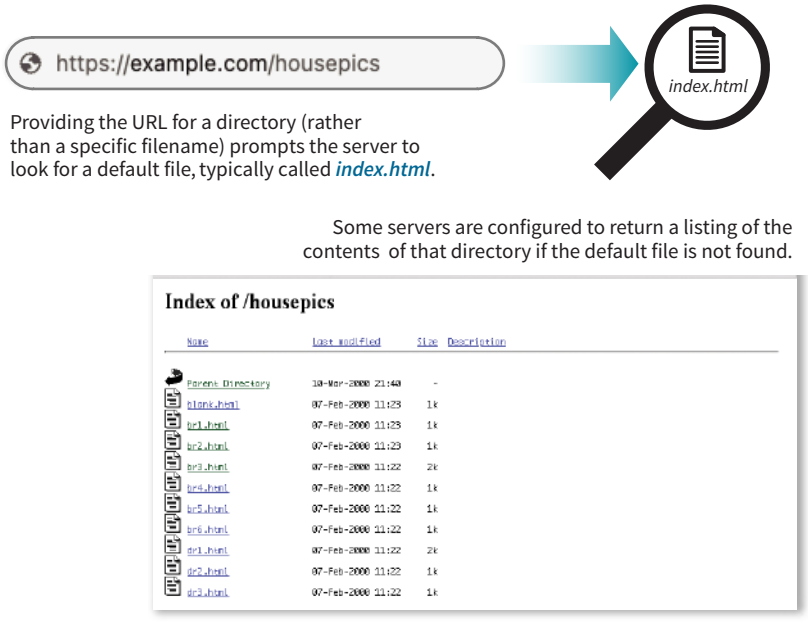


FIGURE 2-2. A server displaying the contents of the directory because the default file is missing

THE ANATOMY OF A WEB PAGE

We're all familiar with what web pages look like in the browser window, but what's happening behind the scenes?

At the top of FIGURE 2-3, you see a minimal web page as it appears in a graphical browser. Although you see it as one coherent page, it is actually assembled from three separate files: an HTML document (*index.html*), a style sheet (*kitchen.css*), and an image (*shelf.jpg*). The HTML document is running the show.

HTML Documents

Adding descriptive tags to a text document is known as "marking up" the document.

You may be surprised to learn that the graphically rich and interactive pages we see on the web are generated by simple, text-only documents. The text file that displays in the browser viewport is referred to as the **source document**.

Take a look at *index.html*, the source document for the Jen's Kitchen web page in FIGURE 2-3. You can see that it contains the text content of the page, plus special **tags** (indicated with angle brackets, < and >) that describe each element on the page. Adding descriptive tags to a text document is known as "marking up" the document. This document has been marked up with tags from the HTML standard, which was created especially for documents with hypertext links (see **Note**).

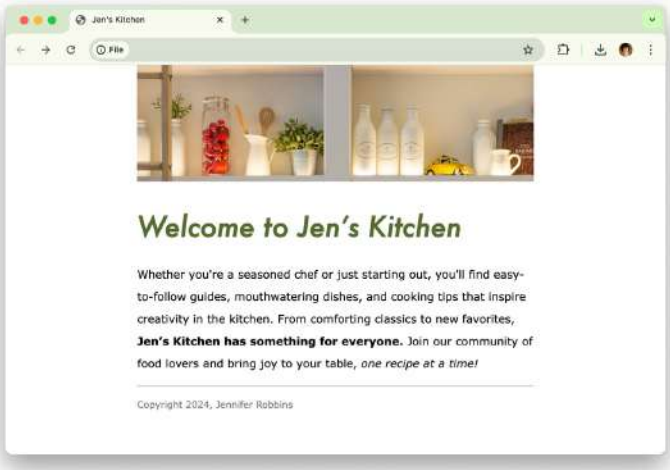
NOTE

There are other markup languages out there that are used to describe specific types of content, such as the parts of math equations (MathML), vector images (SVG), and more.

HTML defines dozens of text elements that make up documents, such as headings, paragraphs, emphasized text, and of course, links. There are also elements that add information about the document (such as its title), media such as images and videos, and widgets for form inputs, just to name a few.

You can view the HTML source document for any web page. EXERCISE 2-1 gives you some prompts and pointers.

There have been several versions of the HTML specification since its birth in 1989. In the past, each specification was locked in and given a version number. Today, the HTML specification is considered a "living standard" and evolves without versioned releases. In your studies, you may run across references to HTML5, which was the last numbered version of HTML maintained by the W3C. It was a big shift in the specification, cleaning up obsolete tags and adding new ones that reflected modern development practices and the more interactive nature of websites. "HTML5" became a marketing buzzword meaning "the latest and greatest web technologies." In this book, when I refer to HTML, I mean the most current, unnumbered standard maintained by the WHATWG. I'll use the term "HTML5" when it is necessary to refer to that particular standard. You can find a complete history of HTML, all its versions, and an overview of what made HTML5 such a breakthrough in my article "From HTML+ to HTML5," available at ***learningwebdesign.com/articles/***.


Jen's Kitchen
File
Welcome to Jen's Kitchen
Whether you're a seasoned chef or just starting out, you'll find easy-to-follow guides, mouthwatering dishes, and cooking tips that inspire creativity in the kitchen. From comforting classics to new favorites, Jen's Kitchen has something for everyone. Join our community of food lovers and bring joy to your table, one recipe at a time!
Copyright 2024, Jennifer Robbins

The web page shown in this browser window consists of three separate files:

- an HTML text document (*index.html*)
- a style sheet (*kitchen.css*)
- one image (*shelf.jpg*)

Tags in the HTML source document give the browsers instructions for how the text is structured and where the image should be placed.

index.html

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Jen's Kitchen</title>
  <link rel="stylesheet" href="kitchen.css">
</head>
<body>

<img src="shelf.jpg" alt="photo of shelves with kitchen items">

<h1>Welcome to Jen's Kitchen</h1>

<p>Whether you're a seasoned chef or just starting out, you'll find easy-to-follow guides, mouthwatering
dishes, and cooking tips that inspire creativity in the kitchen. From comforting classics to new favorites,
<strong>Jen's Kitchen has something for everyone.</strong> Join our community of food lovers and bring joy to
your table, <em>one recipe at a time!</em></p>

<hr>
<small>Copyright 2024, Jennifer Robbins</small>
</body>
</html>
```

kitchen.css

```
body {
  font: normal 1em/2em Verdana, sans-serif; margin: 0 20%; }

h1 { font: italic bold 2.5rem Futura, sans-serif;
  color: darkolivegreen; }

small { color: gray; }
img { width: 100%; height: auto; }
```

shelf.jpg

FIGURE 2-3. The source file, style sheet, and images that make up a simple web page (photo by Jason Leung on Unsplash.com)

EXERCISE 2-1. View source

You can see the HTML file for any web page by viewing its source in a desktop browser. Most modern browsers keep the View Source function with the developer tools and typically open the source document in a separate window, or in a developer's panel at the bottom of the current window.

Here's where to find the View Source function on the major desktop browsers:

Safari: **Develop → Show Page Source**

Chrome: **View → Developer → View Source**

MS Edge: **Tools → Developer → View Source**

Firefox: **Tools → Browser Tools → Page Source**

1. Enter this URL into the browser of your choice:

   `www.learningwebdesign.com/6e/kitchen.html`

   You should see the Jen's Kitchen web page from FIGURE 2-3.
2. Follow the directions for your browser listed above to view the source HTML document for the Jen's Kitchen page. It should be the same as shown in the figure.
3. To view a page that is a little more complicated, take a look at the source for the *learningwebdesign.com* home page.
4. The source for most sites is considerably more complicated than that. View the source of *oreilly.com*. It's got style sheets, scripts, inline SVG graphics…the works! Don't worry if you don't understand what's going on. Much of it will look more familiar by the time you are done with this book.

WARNING

Keep in mind that while learning from others' work is fine, stealing other people's code is poor form (or even illegal). If you want to use code as you see it, ask for permission, and always give credit to those who did the work.

Parsing HTML: "Beads on a String"

Because I was unfamiliar with computer programming when I first began writing HTML in 1993, it helped me to think of the tags and text as "beads on a string" that the browser interprets one by one, in sequence. For example, when the browser encounters an open bracket (`<`), it assumes all of the following characters are part of the tag until it finds the closing bracket (`>`). Similarly, it assumes all of the content following an opening `<h1>` tag is a heading until it encounters the closing `</h1>` tag. This is the manner in which the browser **parses** the HTML document. Understanding the browser's method can be helpful when troubleshooting a misbehaving HTML document.

A Quick Introduction to HTML Markup

You'll be learning the nitty-gritty of markup in **Part II, HTML for Structure**, so I don't want to bog you down with too much detail right now, but there are a few things I'd like to point out about how HTML works and how browsers interpret it.

Read through the HTML document in FIGURE 2-3 and compare it to the browser results. It's easy to see how the elements marked up with HTML tags in the source document correspond to what displays in the browser window.

First, you'll notice that the text characters within brackets (for example, `<body>` and `<strong>`) do not display in the final page. The browser displays only what's *between* the tags—the **content** of the element. The markup is hidden. The tag provides the name of the HTML element—usually an abbreviation, such as "h1" for "heading level 1," or "em" for "emphasized text."

Second, you'll see that most of the HTML tags appear in pairs surrounding the content of the element. `<h1>` indicates that the following text should be a first-level heading; `</h1>` indicates the end of the heading. Some elements, called **void elements** (or **empty elements** in older HTML specs), do not have content. In the Jen's Kitchen source, the `<hr>` tag indicates a void element that tells the browser to "insert a horizontal rule here" as a thematic divider.

Where Are the Pictures?

Obviously, there are no pictures in the HTML file itself, so how do they get there when you view the final page? You can see in FIGURE 2-3 that the image is a separate file. Images are placed in the flow of the text with the HTML image element (`img`), which tells the browser the location of the image (its URL). When the browser sees the `img` element, it makes another request to the server for the image file, then places it in the content flow.

The browser also sends requests to the server for style sheets (like *kitchen.css*), JavaScript files (*.js*), and other embedded media like audio and videos. The browser software (or more specifically, its rendering engine) brings the separate pieces together into the final page.

The assembly of the page generally happens in an instant, so it appears as though the whole page loads all at once. Over slow connections, or if the page includes huge images or media files, the assembly process may be more apparent as images lag behind the text. The page may even need to be redrawn as new images, fonts, and style sheets arrive, although you can write your HTML in a way that prevents this from happening.

Adding a Little Style

Direct your attention to the last key ingredient of our minimal page. Near the top of the HTML document there is a `link` element that points to the style sheet document *kitchen.css*. That style sheet includes a few lines of instructions for how the page should look in the browser. These are style instructions written according to the rules of **Cascading Style Sheets** (**CSS**). CSS allows designers to add visual style instructions (known as the document's **presentation**) to the marked-up text (the document's **structure**, in web design terminology). In **Part III, CSS for Presentation**, you'll get to know the power of Cascading Style Sheets.

FIGURE 2-4 shows the Jen's Kitchen page with and without the style instructions. Browsers come equipped with default styles for every HTML element they support, so if an HTML document lacks custom style instructions, the browser will use its own. For example, the `h1` heading is bigger and bolder than the paragraph text by default. That's what you see in the unstyled page. Even just a few style rules can make big improvements to the appearance of a page.

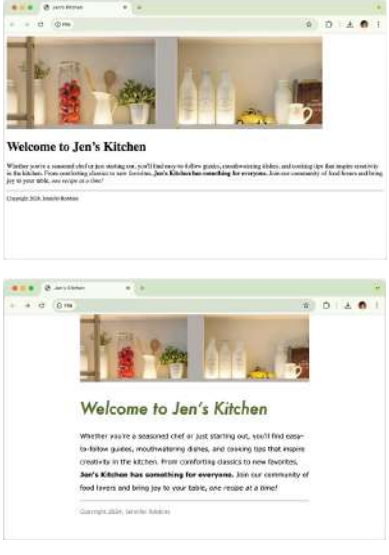


FIGURE 2-4. The Jen's Kitchen page without (top) and with (bottom) custom style rules

Adding Behaviors with JavaScript

To make elements on the page *do* something, you use a scripting language called **JavaScript** (see **Note**). There are no scripts on the Jen's Kitchen page because I thought it best to keep things simple this early in the book, but JavaScript is an essential ingredient in modern websites.

NOTE

JavaScript is not required for the interactivity of links and web forms, which work using HTML alone.

Whereas HTML provides the structure and the CSS style sheet alters how things look, JavaScript adds a behavior component that controls how things *work*. Scripts can be standalone files on the server (with the *.js* suffix) or be written out right in the document. They may be triggered to run immediately when the page loads or be triggered by something the user does, like clicking or hovering on an element or entering something in a form field.

You'll get a basic introduction to JavaScript in **Part IV, JavaScript for Behavior.**

PUTTING IT ALL TOGETHER

To wrap up this introduction to how the web works, let's trace the typical stream of events that occurs with every web page that appears on your screen (FIGURE 2-5):

1. You request a web page, by either typing its URL (for example, `https://learningwebdesign.com/6e/kitchen/`) directly in your browser or clicking a link on a page. The URL contains the information needed to target a specific document on a specific web server on the internet.
2. The browser sends an HTTP request to the server named in the URL and asks for the page. The request also includes information about what languages the user can read and what types of files the browser can accept.
3. The server looks for the requested file and issues an HTTP response in the form of an **HTTP header**. The header includes information about the file, like the last modified date, the length of the file, and its Content-Type (for example, an *.html* file has the Content-Type "text/html").
   a. If the URL specifies a directory (not a file), the server looks for the default file in that directory—in this case *index.html* in the *kitchen* directory.
   b. If the page cannot be found, the server returns an error message. The message typically says "404 Not Found," although more hospitable error messages may be provided. Other error types are possible as well (see the sidebar **"HTTP Status Codes"**).
   c. If the document *is* found, the server retrieves the requested file and returns it to the browser. If the site is dynamic, the server assembles the page from stored data before returning it to the browser.
4. The browser parses the HTML document. If the page contains images (indicated by the HTML `img` element) or other external resources, like scripts or style sheets, the browser contacts the server again to request each resource specified in the markup.
5. The browser inserts each image in the document flow where indicated by the `img` element, applies styles, and runs scripts. And *voilà*! The assembled web page is displayed for your viewing pleasure.

Static Versus Dynamic Sites

Static websites consist of HTML files with fixed content that display the same information to every visitor. In other words, each page you see in the browser is a view of a single HTML file on the server. This book focuses on the creation of static web pages, as they are straightforward and the best starting place for beginners.

By contrast, **dynamic websites** are generated with backend programming such as PHP, Node.js, Ruby, or ASP.NET. Each page is generated by the application on the fly by pouring content and data from a database into a page template. The final pages may be customized for each user. For extremely large sites with hundreds or thousands of pages, setting up and maintaining a dynamic site is considerably less work than creating and storing every page as a static HTML document individually.

HTTP Status Codes

Servers issue status codes in response to browser requests. The full list of status codes is quite long (you can read about them all at *en.wikipedia.org/wiki/List_of_HTTP_status_codes*), but here are a few common responses:

200	OK
301	Moved Permanently
302	Moved Temporarily
404	Not Found
500	Internal Server Error

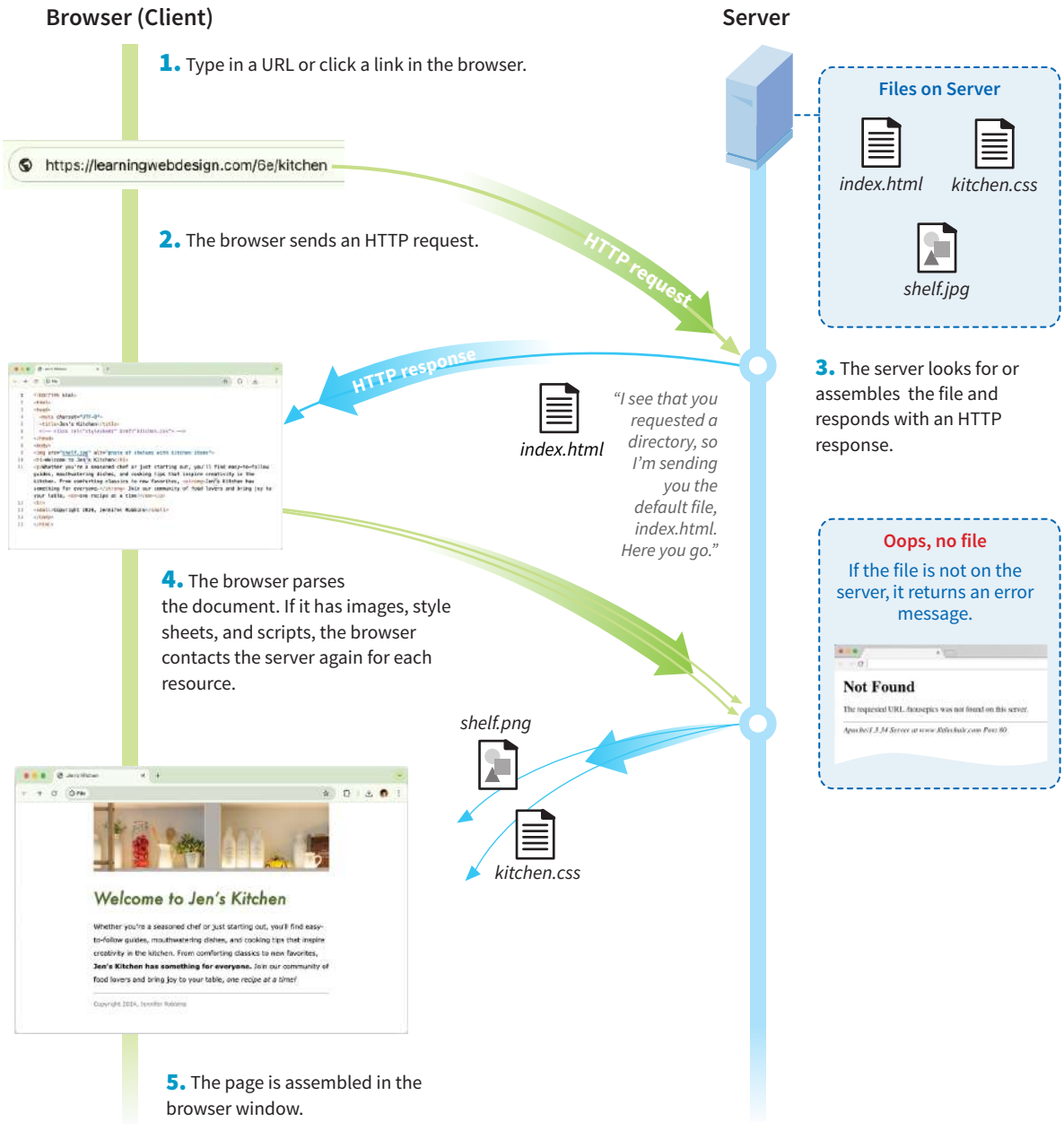

Browser (Client)
Server
1. Type in a URL or click a link in the browser.
https://learningwebdesign.com/6e/kitchen
2. The browser sends an HTTP request.
HTTP request
HTTP response
Files on Server
index.html
kitchen.css
shelf.jpg
3. The server looks for or assembles the file and responds with an HTTP response.
index.html
"I see that you requested a directory, so I'm sending you the default file, index.html. Here you go."
Oops, no file
If the file is not on the server, it returns an error message.
Not Found
4. The browser parses the document. If it has images, style sheets, and scripts, the browser contacts the server again for each resource.
shelf.png
kitchen.css
Welcome to Jen's Kitchen
5. The page is assembled in the browser window.

FIGURE 2-5. How browsers display web pages

I should note that I've depicted a traditional and simplified scenario here to tell you how web pages are put together. These days, it is common for web pages to be generated by **content management systems** (**CMSs**) that keep content in databases and use templates to assemble the data into pages on the fly. In that case, in step 3b, there is a more complicated process of assembling the file from various parts rather than just handing off an existing file.

We're at the end of the chapter, so I'd say it's time for a little quiz!

TEST YOURSELF

Getting Your Pages on the Web

If you would like more information about registering domain names and finding a server to host your site, see my article **"Getting Your Pages on the Web"** at *learningwebdesign.com/articles/*.

Let's play a round of "Identify That Acronym!" The following are a few basic web terms mentioned in this chapter. Can you match them to their descriptions? Answers are in **Appendix A**.

1.	HTML _____	a.	Home of Mosaic, the first graphical browser
2.	W3C _____	b.	The location of a web document or resource
3.	CERN _____	c.	The markup language used to describe web content
4.	CSS _____	d.	Matches domain names with numeric IP addresses
5.	HTTPS _____	e.	A protocol for file transfer
6.	IP _____	f.	A protocol for transferring web documents on the internet securely
7.	URL _____	g.	The language used to specify how web content looks
8.	NCSA _____	h.	The particle physics lab where the web was born
9.	DNS _____	i.	Internet Protocol
10.	FTP _____	j.	The organization that monitors web technologies

CHAPTER
3

SOME BIG CONCEPTS YOU NEED TO KNOW

IN THIS CHAPTER

The web on mobile devices

The benefits of web standards

Progressive enhancement

Responsive web design

Accessibility

Site performance

Before we get to the nuts and bolts of HTML, CSS, and JavaScript, I want to introduce some "Big Concepts" that every web designer needs to know. I'll be referring to the terminology introduced in this chapter frequently, so it's not to be missed.

The heart of the matter is that as web designers, we never know exactly how the pages we create will be viewed. We don't know which of the dozens of browsers might be used, whether that browser is on a desktop computer or something more portable, how large the viewport will be, what fonts are installed, whether functionality such as JavaScript is enabled, how fast the internet connection is, whether the pages are being read by a screen reader, and so on. The Big Concepts in this chapter are primarily reactions to—and methods for—coping with the inescapable element of *the unknown* in our medium. They include the following:

- The multitude of devices
- Web standards
- Progressive enhancement
- Responsive web design
- Accessibility
- Site performance

Because we're just getting started, I'll keep the descriptions brief and fairly nontechnical. The goal is that you have a basic understanding of what I mean by terms like "progressive enhancement" when you encounter them in lessons later. Many excellent articles and books have been written on each of these topics and their related production techniques, and I'll provide pointers to resources for further reading.

A MULTITUDE OF DEVICES

Until 2007, web designers could be relatively certain that our users were visiting our sites while sitting at their desks, looking at a large monitor, using a speedy internet connection. We had all more or less settled on 960 pixels as a good width for a web page, based on the most common monitor size. Back then, our biggest concern was dealing with the dozen or so desktop browsers and jumping through a few extra hoops to support quirky old versions of Internet Explorer (it was no picnic!).

Although people could access web pages and web content on mobile phones prior to 2007, the introduction of the iPhone and Android smartphones, as well as faster networks, heralded a huge shift in how, when, and where we do our web surfing. Since then, we've seen the introduction of phones and tablets of all different dimensions, as well as web browsers on TVs, gaming systems, and other devices. And the diversity is only going to increase. I think mobile web design expert Brad Frost (***bradfrost.com***) sums it up nicely in his illustrations in FIGURE 3-1.

Resist making assumptions about network speed and context based on the screen size.

The challenge of designing for all of these devices goes beyond addressing differing screen sizes. There is a world of difference between using a site over a broadband connection and over a slow cell network. Designers need to resist making assumptions about network speed and context based on the screen size. Just because it's a small screen doesn't mean it's a slow connection or that the person is in a hurry—it's not uncommon to leisurely browse the web on a smartphone while sitting on the couch at home with a solid WiFi connection. Conversely, iPads with larger, high-resolution displays may be accessing the internet on slower cell connections. In other words, it's complicated!

For a lot of sites today, more people are visiting via their mobile devices than on a desktop computer. Already, a significant portion of Americans use their mobile phones as their *only* means of access to the internet. That means it's critical to get the design and functionality right. We've made huge strides in

FIGURE 3-1. Brad Frost sums up the reality of device diversity nicely.

serving a pleasing experience to users with handheld devices, and the technology for targeting their needs continues to head in the right direction.

What I want you to learn here is that the way you see your design as you're working on it on your nice desktop machine is not how it will be experienced by everyone. Some will see it much smaller. Some will see it load painfully slowly. Some may be looking at it on a TV across the room. Some may be listening to it with a screen reader. All web design professionals should keep this fact in mind.

Further Reading

- *Mobile First* by Luke Wroblewski (A Book Apart). Luke was way ahead of the curve in insisting that sites work well on mobile devices. Published in 2011, it's now more of a historical record, but the concepts are still useful today. You can read it for free online at ***mobile-first.abookapart.com***.
- The book recommendations for responsive web design, accessiblity, and performance that I make later in this chapter are also relevant to delivering satisfying experiences on mobile devices.

STICKING WITH THE STANDARDS

So how do we deal with this diversity? A good start is to follow the standards documented by the World Wide Web Consortium. Sticking with web standards is your primary tool for ensuring your site is consistent on all standards-compliant browsers (that's virtually all browsers in current use). It also helps make your content forward-compatible as web technologies and browser capabilities evolve.

Sticking with web standards is your primary tool for ensuring your site is as consistent as possible.

The notion of standards compliance may seem like a no-brainer, but it used to be that everyone, including the browser makers, played fast and loose with HTML and scripting. The price we paid was incompatible browser implementations and the need to create multiple versions of sites to make them work for everyone. I'll talk more about web standards throughout this book, so I won't go into too much detail here. Suffice it to say that the web standards are your friends.

Further Reading

- The W3C (***w3.org/standards***) and WHATWG (***whatwg.org***) sites are the primary resources for all web standards documents.
- *Designing with Web Standards*, 3rd edition, by Jeffrey Zeldman (New Riders) is the bible for standards compliance and how it makes good business sense. It's getting on in years, but the fundamentals are still solid.

PROGRESSIVE ENHANCEMENT

Progressive enhancement is a strategy for coping with unknown browser capabilities.

With a multitude of browsers comes a multitude of levels of support for the web standards. In fact, no browser has fully implemented all the standards, and there are always emerging technologies steadily gaining steam. Furthermore, users can set their own browser preferences, so, for example, they may have a browser that supports JavaScript but have chosen to turn it off. The point here is that we are faced with a wide range of browser capabilities, from basic HTML support to all the bells and whistles.

Progressive enhancement is one strategy for dealing with unknown browser capabilities (see **Note**). The first rule of progressive enhancement is to start with a baseline experience that makes the content or core functionality available to even the most rudimentary browsers or assistive devices. From there, you layer on more advanced features for the browsers that can handle them. You might finish with some "nice to have" features, like using drop caps (enlarging the first letter of text and letting the text lines wrap around it) or a cutting-edge image format like AVIF, that enhance the experience for users with the most advanced browsers.

Progressive enhancement is an approach that can be applied to all components of page design and production, including authoring in HTML, styling content with CSS, and adding scripted behaviors with JavaScript:

Authoring strategy

When an HTML document is written in a logical order and its elements are marked up in a meaningful way, it will be usable on the widest range of browsing environments, including the oldest browsers, future browsers, and mobile and assistive devices. It may not look exactly the same, but the important thing is that your content is available. This approach also ensures that search engines like Google will catalog the content correctly. A clean HTML document with its elements accurately and thoroughly described is the foundation of accessibility.

Styling strategy

You can create layers of experience by taking advantage of the way browsers parse style sheet rules. The knowledge that browsers simply ignore properties and rules they don't understand gives you license to innovate without breaking things on older browsers. You just have to be mindful of styling the baseline experience first, then adding improvements once the minimum requirements are met.

Scripting strategy

Although it's not as common as it used to be, users can turn off JavaScript in their browsers. There is also a small chance that an external *.js* file won't load. Use progressive enhancement to make sure basic functionality—such as linking from page to page or accomplishing essential tasks

NOTE

The shift to using JavaScript frameworks that render page components on the server has taken some of the responsibility for standards support off of browsers. For that reason, in the current professional web development landscape, progressive enhancement is less in the forefront of developers' minds than it was a decade or so ago. However, if you are coding a site traditionally, with HTML and CSS (and not a framework), accommodating older browsers with progressive enhancement is still a useful approach. This is why I have chosen to include it as one of the key principles.

like data submission via forms—is intact even when JavaScript is off. In this way, you ensure the integrity of the baseline experience, and enhance it when JavaScript is available.

Further Reading

- *Adaptive Web Design: Crafting Rich Experiences with Progressive Enhancement*, 2nd edition, by Aaron Gustafson (New Riders). There's no better introduction to the progressive enhancement approach than this book. Aaron is also the author of the chapters in **Part IV, JavaScript for Behavior**.
- *The Uncertain Web: Web Development in a Changing Landscape* by Rob Larson (O'Reilly).
- *Resilient Web Design* by Jeremy Keith (online, self-published).

RESPONSIVE WEB DESIGN

By default, most browsers on small devices such as smartphones and tablets shrink a web page down to fit the screen and provide mechanisms for zooming and moving around the page. Although this technically works, it's not a great experience. The text is too small to read, the links are too small to tap, and all that zooming and panning around is distracting.

Responsive web design (**RWD**) is a strategy for providing appropriate layouts to devices based on the size of the viewport (the browser window on a desktop computer, or a full-screen browser application on a mobile device). The key to RWD is serving a single HTML document at one URL to all devices, but applying different styles based on the viewport size in order to provide the most optimized layout for each device. For example, when the page is viewed on a smartphone, it appears in one column with large links for easy tapping. But when that same page is viewed on a large desktop browser, the content rearranges into multiple columns with traditional navigation elements. It's like *magic*! (Except that it's actually just CSS.)

Responsive web design is a strategy for dealing with unknown screen size.

The web design community has been abuzz about responsive design since Ethan Marcotte first wrote about it and coined the phrase in his article "Responsive Web Design" on *A List Apart* in 2010 (***alistapart.com/articles/responsive-web-design/***).

FIGURE 3-2 shows some examples of responsive sites at the typical dimensions for a smartphone, tablet, and desktop monitor. Try opening one of the responsive sites in your desktop browser and then resizing the window, making it narrower and wider. Watch as the layout changes based on the window size. *Très* cool.

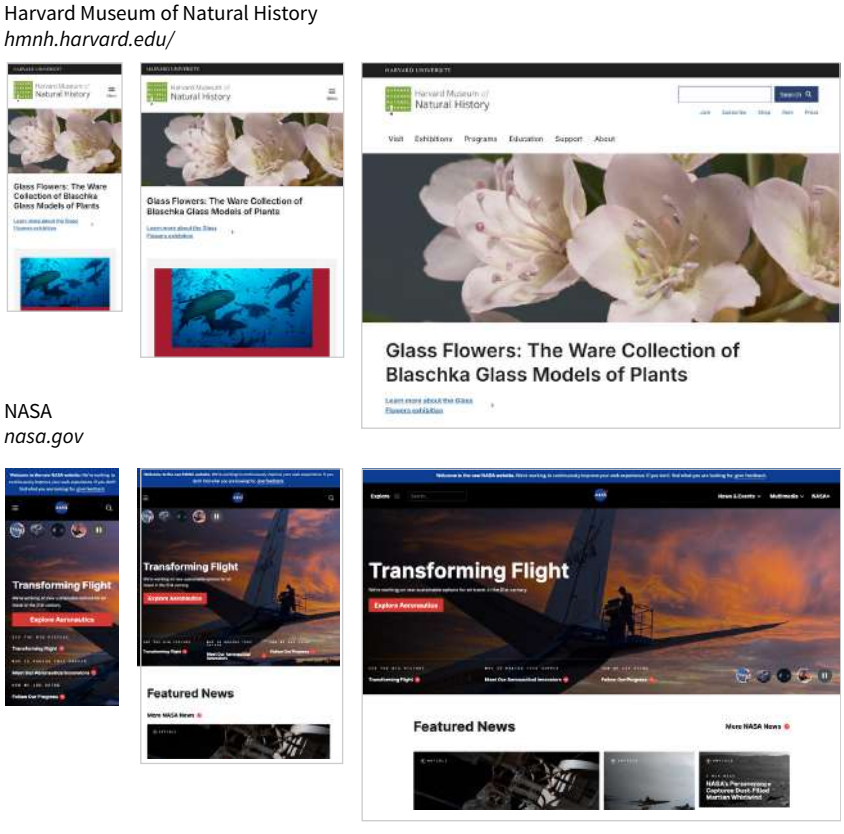


FIGURE 3-2. A responsive site's layout changes based on the size of the browser window

Responsive web design helps with matters of layout at different viewport sizes, but it's not a solution to all mobile web design challenges. The fact is that providing the best experiences for your users and their chosen devices may require optimizations that go beyond adjusting the look and feel. You can better address some problems by having the server detect the device and its capabilities and send back appropriately formatted content.

For some sites and services, it may be preferable to build a separate mobile site (see the **"M-Dot Sites"** sidebar) with a customized interface and feature set that takes advantage of phone capabilities like geolocation. That said, although responsive design won't fix everything, it has become the standard method for building a website and delivering satisfactory experiences on a wide variety of browsers.

Further Reading

I'll cover responsive web design in more detail in **Chapter 19, Responsive Web Design**, once you have more code experience under your belt. You'll find plenty of resources there to continue your responsive design education.

M-Dot Sites

Some companies and services choose to build an entirely separate site, with a unique URL, just for mobile devices. M-dot sites (so called because their URLs typically begin with "m." or "mobile.") usually offer a reduced set of options and may also include mobile-specific features such as geolocation. A dedicated mobile site may be the best solution if you know that your mobile users have very different usage patterns than folks seated at a desk. Facebook, for example, delivers a streamlined interface to mobile browsers with *m.facebook.com*.

The downside of creating a dedicated mobile site is that it's more than twice the work. It requires additional content planning, design templates, production time, and ongoing maintenance. In general, m-dot sites have faded out of use in favor of RWD. Google has helped to speed this process along by switching to "mobile-first" indexing (*developers.google.com/search/docs/crawling-indexing/mobile/mobile-sites-mobile-first-indexing*), basing search results on the contents of a site as it appears on a mobile device. If search result rankings are a concern, you will get more mileage from going responsive.

ACCESSIBILITY

I've talked a bit about the daunting number of browsers in use today, but so far, I've only addressed visual browsers controlled with mouse pointers or fingertips. It is critical, however, to keep in mind that people access the web in many different ways—with a keyboard, mouse, voice commands, screen readers, Braille output, magnifiers, joysticks, foot pedals, and so on. Web designers must build pages in a manner that creates as few barriers as possible to getting to information, regardless of the user's ability and the device used to access the web. In other words, you must design for **accessibility**.

Although intended for users with disabilities such as poor vision or limited mobility, the techniques developed for accessibility also benefit other users with less-than-optimum browsing experiences. Accessible sites are also more effectively indexed by search engines such as Google. There are four broad categories of disabilities that affect how people interact with their computers and the information on them:

Vision impairment

People with low or no vision may use an assistive device such as a screen reader (see **Note**), Braille display, or screen magnifier to get content from the screen. Low-vision site visitors may also simply use the browser's text zoom function to make the text large enough to read.

Mobility impairment

Users with limited or no use of their hands may use special devices such as modified mice and keyboards, foot pedals, voice commands, or joysticks to navigate the web and enter information.

Auditory impairment

Users with limited or no hearing will miss out on audio aspects of multimedia, so it is necessary to provide alternatives, such as transcripts for audio tracks or captions for video.

Cognitive impairment

Users with memory, reading comprehension, problem solving, and attention limitations benefit when sites are designed simply and clearly. These qualities are helpful to anyone using your site.

The W3C started the **Web Accessibility Initiative** (**WAI**) to address the need to make the web usable for everyone. The WAI site (***w3.org/WAI***) is an excellent starting point for learning more about web accessibility. One of the documents produced by the WAI to help developers create accessible sites is the Web Content Accessibility Guidelines (WCAG). You can read about the different versions at ***w3.org/WAI/standards-guidelines/wcag/***. The US government based its Section 508 accessibility guidelines on the Priority 1 points of WCAG (see the sidebar **"Government Accessibility Requirements: Section 508"**). All sites benefit from these guidelines, but if you are designing a government site, adherence is a requirement.

TERMINOLOGY

A11y is a common abbreviation for Accessibility that provides the first (A) and last letter (y) and counts the 11 letters in between.

NOTE

*It's not a bad idea to get your own screen reader to get familiar with how your site will be experienced when read aloud. Windows users can download the NVDA screen reader (**nvaccess.org/download/**) for free. Mac users can use VoiceOver, the screen reader that is built into the operating system.*

Another W3C effort is the **WAI-ARIA** (**Accessible Rich Internet Applications**) spec, which addresses the accessibility of web applications that include dynamically generated content, scripting, and advanced interface elements that are particularly confounding to assistive devices. The ARIA recommendation defines a number of roles for content and widgets that web developers can explicitly apply using the `role` attribute, including include `menubar`, `progressbar`, `slider`, `timer`, and `tooltip`, to name just a few. For the complete list of roles, go to ***w3.org/TR/wai-aria/#role_definitions***.

Further Reading

The following resources are good starting points for further exploration of web accessibility:

- The Web Accessibility Initiative (WAI), ***w3.org/WAI/***
- WebAIM: Web Accessibility in Mind, ***webaim.org***
- Accessible Web, ***accessibleweb.com*** (This company provides accessibility audits and scanning software; there is a fee for their services, but they offer a free trial version.)
- *Accessibility Handbook: Making 508 Compliant Websites* by Katie Cunningham (O'Reilly)

US Government Accessibility Requirements: Section 508

If you create a site that receives federal funding from the US government, you are required by law to comply with the Section 508 guidelines, which ensure that electronic information and technology are available to people with disabilities. State and other publicly funded sites may also be required to comply.

The following guidelines, excerpted from the Section 508 Standards at ***access-board.gov/ict/***, provide a good checklist for basic accessibility for all websites:

1. A text equivalent for non-text elements shall be provided (e.g., via the "alt" attribute or in element content).
2. Equivalent alternatives for any multimedia presentation shall be synchronized with the presentation.
3. Web pages shall be designed so that all information conveyed with color is also available without color—for example, from context or markup.
4. Documents shall be organized so they are readable without requiring an associated style sheet.
5. Row and column headers shall be identified for data tables.
6. Markup shall be used to associate data and header cells for tables with two or more levels of row or column headers.
7. Pages shall be designed to avoid causing the screen to flicker with a frequency greater than 2 Hz and lower than 55 Hz.
8. When pages utilize scripting languages to display content, or to create interface elements, the information provided by the script shall be identified with functional text that can be read by assistive technology.
9. When a web page requires that an applet, plug-in, or other application be present on the client system to interpret page content, the page must provide a link to a plug-in or applet that complies with §1194.21(a) through (l).
10. When electronic forms are designed to be completed online, the form shall allow people using assistive technology to access the information, field elements, and functionality required for completion and submission of the form, including all directions and cues.
11. A method shall be provided that permits users to skip repetitive navigation links.
12. When a timed response is required, the user shall be alerted and given sufficient time to indicate more time is required.

Inclusive Design

Accessibility generally addresses physical limitations—compromised sight, hearing, mobility, etc.—that might prevent users from accomplishing their goals. It is one part of a broader design methodology, **inclusive design**.

It is natural to subconsciously assume your users are like you or do things the way you do. The key to inclusive design is making sure that you include varying perspectives as part of the design and problem-solving process. Or, as practitioners of inclusive design put it: "Design with, not for."

Inclusive design takes into account all the ways that people are unique so that nobody feels unwelcome or left out. It may address age, economic situation, neurodiversity, location, language, culture, race, accessibility, and more.

Even small adjustments can make a site feel more inclusive. For example, a person of mixed race may feel excluded while filling out a form if an input forces them to choose a single race. A "mixed race" option or the ability to choose multiple options would offer less friction and make them feel more welcome.

When people feel like your site or product is not quite for them, it affects the bottom line, which is why inclusive design processes have become a priority in web design. It's good business. You can learn more with these resources:

- *Mismatch: How Inclusion Shapes Design* by Kat Holmes (The MIT Press)
- *Inclusive Design Communities* by Sameera Kapila (***samkapila.com/books/***)
- Microsoft's Inclusive Design resources, including articles and guidebooks (***inclusive.microsoft.design***)
- "Inclusive Design," an overview article by Alita Joyce of Nielson Norman Group (***nngroup.com/articles/inclusive-design***)
- The Neurodiversity Design System (***neurodiversity.design***), design patterns for accommodating a range of neurodiversity

SITE PERFORMANCE

While dial-up connections are mostly a thing of the past, network conditions still vary dramatically by connection type, location, and even time of day. Devices, too, vary dramatically in how much computational power and memory they have available. Regardless of who you are or where you live, we've all felt the frustration of a slow-loading site at some point or another.

Site performance factors into search engine rankings, and it has repeatedly been shown to have a noticeable impact on how people use sites. In 2006, Amazon.com showed that reducing page load times by just 100 milliseconds (ms) resulted in a 1% increase in revenue (interestingly, Walmart found the exact same impact in its own study).* A study by Google in 2009† showed that the addition of just 100 to 400 ms to the load time of their search results page resulted in reduced searches (–0.2 to –0.6%). Other studies have shown that users expect a site to load in under 2 seconds, and nearly a third of your audience will leave your site for another if it doesn't. Furthermore, those people aren't likely to come back. Google has added site speed to its search algorithm, so if your site is a slowpoke, it's not likely to show up in that coveted first screen of results. The takeaway here is that site performance (down to the millisecond!) matters a lot.

TIP

You can find an ever-growing collection of case studies showing the business impact of performance at ***wpostats.com***, curated by Tim and Tammy Everts.

* Statistic from "Make Data Matter," PowerPoint presentation by Greg Linden of Stanford University (2006).

† "Speed Matters," ***googleresearch.blogspot.com/2009/06/speed-matters.html***.

There are many things you can do to improve the performance of your site. Quite a few of them fall beyond the scope of this book, but a solid starting point is to focus on limiting file sizes and reducing the number of requests to the server. The following list only scratches the surface for site optimization, but it gives you a general idea of what can be done:

- Optimize images so they are the smallest file size possible without sacrificing quality. You'll learn image optimization techniques in **Chapter 27, Image Production Techniques**.
- Streamline HTML markup, avoiding unnecessary levels of nested elements.
- Minimize the size of HTML and CSS documents by removing extra character spaces and line returns.
- Keep JavaScript to a minimum.
- Add scripts in such a way that they load in parallel with other page assets and don't block rendering.
- Don't load unnecessary assets (such as images, scripts, or JavaScript libraries).
- Reduce the number of times the browser makes requests of the server (known as **HTTP requests**).

Every trip to the server in the form of an HTTP request takes a few milliseconds, and those milliseconds can add up. All those little Instagram widgets, Facebook Like buttons, and advertisements can make dozens of server requests each.

You may be surprised to see how many server requests even a simple site makes. If you'd like to see for yourself, you can use the Network tool available with the Developer tools in Chrome, Safari, or Firefox. The Network tool displays each request to the server and how many milliseconds it took. It works similarly in all the browsers, but here's how you use it in Chrome:

1. Launch the browser and go to any web page.
2. Go to the View menu and select Developer → Developer Tools. A panel will open at the bottom of the browser. If you don't see it, it may be on the right side, but you can change its position with the "Dock Side" setting under the options icon (three stacked dots).
3. Select the Network tab in the tools view. The chart (commonly referred to as a **waterfall chart**) shows you all the requests made and assets downloaded. The columns on the right show the amount of time each request took, in milliseconds. At the bottom of the chart, you can see a summary of the number of requests made and the total amount of data transferred.

Testing Performance Using Lighthouse

Google's Lighthouse is a great starting point for testing your site's performance and seeing what sort of improvements can be made. You can run Lighthouse as a standalone tool, from within Chrome's developer tools, or through Google's PageSpeed Insights (*pagespeed.web.dev*).

FIGURE 3-3 shows the performance waterfall chart for *learningwebdesign.com*. You can poke around any site on the web this way. It can be very educational.

I won't address site performance in deep technical detail in this book, but I do want you to remember the importance of keeping file sizes as small as possible and eliminating unnecessary server requests in your web design work.

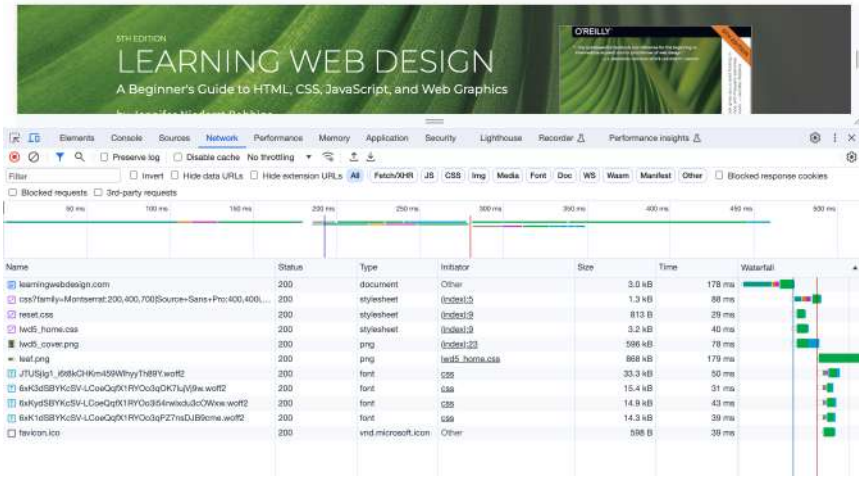


FIGURE 3-3. Waterfall charts such as this one created by the Chrome Network developer tool show the individual server requests made by a web page and the amount of time each request takes.

Further Reading

There are other techniques that are too technical for this book, and I figure if you're reading this book, you're probably not quite ready to become a site performance wizard. But when you are ready to take on this challenge, here are some resources that should help:

- Lara Hogan has assembled a list of performance-related studies, tools, and resources at *larahogan.me/design*. You can also read her book *Designing for Performance* (O'Reilly) there for free.
- *High Performance Mobile Web: Best Practices for Optimizing Mobile Web Apps* by Maximiliano Firtman (O'Reilly) covers optimization methods and tools to check your progress.
- *Web Performance in Action: Building Fast Web Pages* by Jeremy Wagner (Manning) is a fantastic introduction to performance optimization techniques.

TEST YOURSELF

Here are a few questions that check your knowledge of the Big Concepts covered in this chapter. If you're stumped, you can find the answers in **Appendix A**.

1. List at least two unknown factors you need to consider when designing and developing a site.

2. Match the technology or practice on the left with the problem it best addresses on the right:

1. ____ Progressive enhancement	a. Assistive reading and input devices
2. ____ Responsive design	b. Slow connection speeds
3. ____ WAI-ARIA	c. All levels of browser capabilities
4. ____ Site performance optimization	d. A variety of screen sizes

3. Web accessibility strategies take into account four broad categories of disabilities. Name at least three, and provide a measure you might take to ensure content is accessible for users with each type of disability.

4. Why might you use a waterfall chart?

II

HTML FOR STRUCTURE

CHAPTER
4

CREATING A SIMPLE PAGE

HTML OVERVIEW

IN THIS CHAPTER

An introduction to elements and attributes

Marking up a simple web page

The elements that provide document structure

Troubleshooting broken web pages

Part I, Getting Started, provided a general overview of the web design environment. Now that we've covered the Big Concepts, it's time to roll up our sleeves and start creating a real web page. It will be an extremely simple page, but even the most complicated pages are based on the principles described here.

In this chapter, we'll create a web page step-by-step so you can get a feel for what it's like to mark up a document with HTML tags and apply styles. The exercises allow you to work along.

In this chapter you will:

- Get a feel for how markup works, including an understanding of elements and attributes
- See how browsers interpret HTML documents
- Learn how HTML documents are structured
- Get a first glimpse of a style sheet in action

Don't worry about learning the specific text elements or style sheet rules at this point; we'll get to those in the following chapters. For now, just pay attention to the process, the overall structure of the document, and the new terminology.

A WEB PAGE, STEP-BY-STEP

You got a look at an HTML document in **Chapter 2, How the Web Works**, but now you'll get to create one yourself and play around with it in the browser. You'll start with a page of raw text for the fictional Black Goose Bistro site and turn it into a styled web page in five steps that cover the basics of page production:

Step 1: Start with content. As a starting point, we'll write up some raw text content and see what browsers do with it.

Step 2: Give the document structure. You'll learn about HTML element syntax and the elements that set up areas for metadata and page content.

Step 3: Identify text elements. You'll describe the content using the appropriate text elements and learn about the proper way to use HTML.

Step 4: Add an image. By adding an image to the page, you'll learn about attributes and void elements.

Step 5: Change how the text looks with a style sheet. This exercise gives you a taste of formatting content with Cascading Style Sheets.

By the time we're finished, you'll have written the document for the page shown in FIGURE 4-1. It's not very fancy, but you have to start somewhere!

BLACK GOOSE BISTRO
The Restaurant
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.
Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.
Location and Hours
Seekonk, Massachusetts;
Monday through Thursday 11am to 9pm;
Friday and Saturday, 11am to midnight

FIGURE 4-1. We'll write the HTML document for this page in five steps.

We'll be checking our work in a browser frequently throughout this demonstration—probably more than you would in real life, but because this is an introduction to HTML, it's helpful to see the cause and effect of each small change to the source file along the way. Now would be a good time to get set up with a text or code editor, so you can follow along with the exercises.

LAUNCH A TEXT EDITOR

Throughout the book, we'll be writing out HTML documents by hand, so the first thing you'll need to do is launch a text or code editor.

The text editor that is provided with your operating system, such as Notepad (Windows) or TextEdit (Mac), will do for these purposes. The advantages to using a plain text editor are that you have it already, and, more importantly, it won't help you code (like automatically closing element tags for you, for example), so you'll have to be more mindful when writing.

If you would like to use a code editor, like Visual Studio Code or others I talked about in **Chapter 1, Teams and Tools**, I'm fine with that, too. It will give you a more realistic web developer experience. At this point, I do recommend that you avoid any sort of visual drag-and-drop editor, like Dreamweaver, Wix, or similar. I want you to get a feel for marking up a document manually (see the sidebar **"HTML the Hard Way"**).

This section shows how to create HTML documents in Notepad and TextEdit. Even if you've used these programs before, skim through for some special settings that will make the exercises go more smoothly. I'll start with Notepad; Mac users can jump ahead. You can also skip ahead if you would prefer to use a dedicated code editor.

Creating a New Document in Notepad (Windows)

These are the steps to creating a new document in Notepad on Windows 11 (FIGURE 4-2):

1. Search for "Notepad" to access it quickly. Click on Notepad to open a new document window, and you're ready to start typing [**1**].
2. Next, make the extensions visible. This step is not required to make HTML documents, but it will help make the file types clearer at a glance.
3. Open the File Explorer. Select "View" from the toolbar [**2**], then select "Show," and be sure there is a check mark next to "File Name Extensions" [**3**]. The file extensions will now be visible.

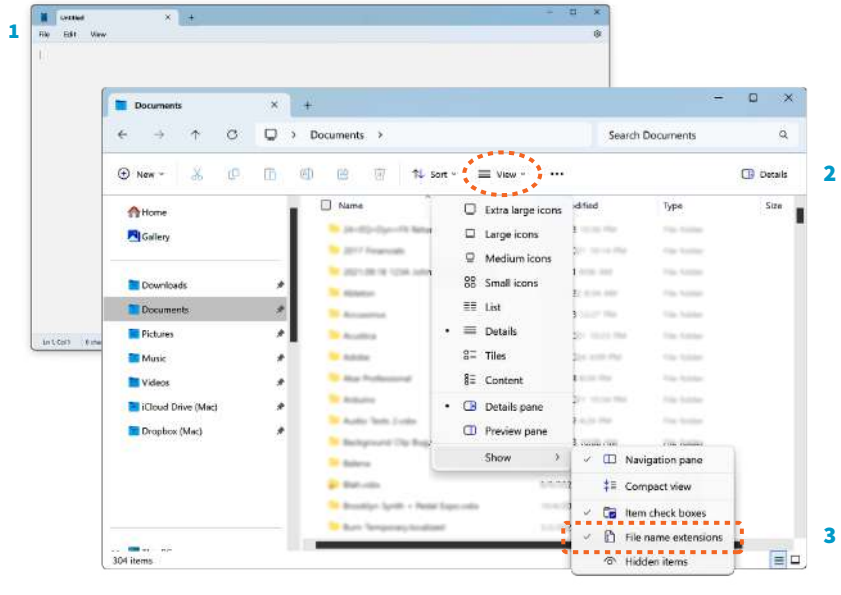


FIGURE 4-2. Creating a new document in Notepad

HTML the Hard Way

I stand by the method of teaching HTML the old-fashioned way—*by hand*. There's no better way to truly understand how markup works than typing it out, one tag at a time, and then opening your page in a browser. It doesn't take long to develop a feel for marking up documents properly.

Although you may choose to use a visual or drag-and-drop web authoring tool or service down the line, understanding HTML will make using that tool easier and more efficient. In addition, you'll be glad that you can look at a source file and understand what you're seeing. This is also crucial for troubleshooting broken pages or fine-tuning the default formatting that some web tools produce.

And, for what it's worth, professional web developers tend to mark up content manually for better control over the code and the ability to make deliberate decisions about what elements to use.

Creating a New Document in TextEdit (macOS)

By default, TextEdit creates **rich-text** documents—that is, documents that have hidden style-formatting instructions for setting font size, weight, and so on. You can tell that TextEdit is in rich-text mode when it has a formatting toolbar at the top of the window. HTML documents need to be plain-text documents, so we'll need to change the format, as shown in this example (FIGURE 4-3):

1. Use the Finder to look in the *Applications* folder for TextEdit. When you've found it, double-click the name or icon to launch the application.
2. In the initial TextEdit dialog box, click the New Document button in the bottom-left corner. If you see the text formatting menu at the top of the Untitled document, you are in rich-text mode [**1**]. If you don't, you are in plain-text mode [**2**]. Either way, there are some preferences you need to set.
3. Close that document, and open the Preferences dialog box from the TextEdit menu.
4. Change these preferences:
   - On the New Document tab, select "Plain text" [**3**]. Under Options, deselect all of the automatic formatting options except "Smart quotes and dashes in rich text documents only" [**4**].
   - On the "Open and Save" tab, select "Display HTML files as HTML code instead of formatted text" [**5**] and deselect "Add '.txt' extension to plain text files" [**6**]. The rest of the defaults should be fine.
5. When you are done, click the red button in the top-left corner to close the dialog box.
6. Now create a new document by selecting File → New. The formatting menu will no longer be there, and you can save your text as an HTML document. You can always convert a document back to rich text by selecting Format → Make Rich Text when you are not using TextEdit for HTML.

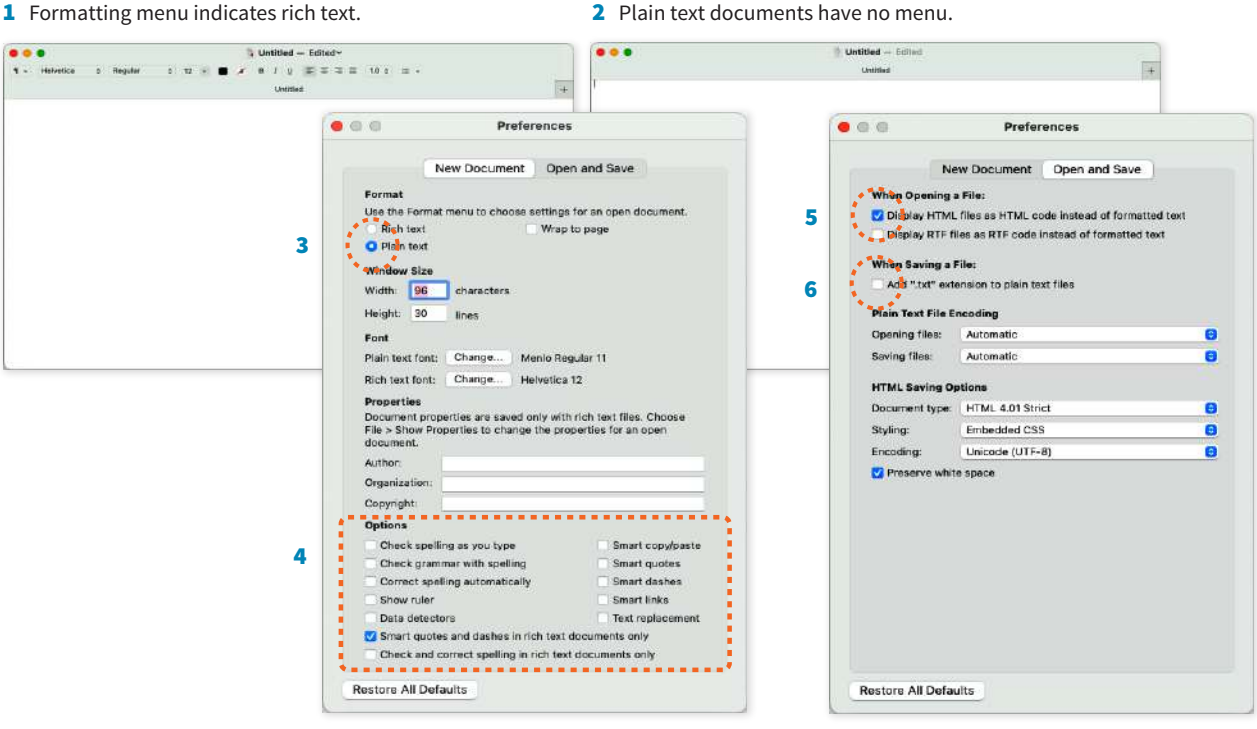


FIGURE 4-3. Launching TextEdit in macOS and choosing "Plain text" settings in the Preferences

STEP 1: START WITH CONTENT

Now that we have our new document, it's time to get typing. A web page is all about content, so that's where we begin our demonstration. EXERCISE 4-1 walks you through entering the raw text content and saving the document in a new folder.

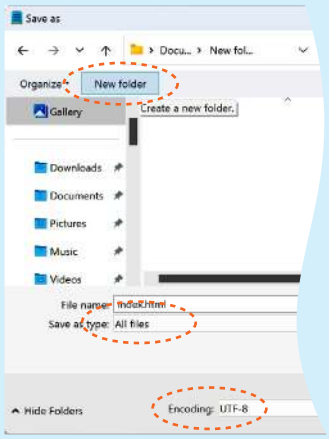
EXERCISE 4-1. Entering content

1. Type the home page content below into the new document in your text editor. Copy it exactly as you see it here, keeping the line breaks the same for the sake of playing along. The raw text for this exercise is also available online at *learningwebdesign.com/6e/materials/*.

```
Black Goose Bistro

The Restaurant
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed
atmosphere. The menu changes regularly to highlight the freshest local
ingredients.

Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle
events from snacks for a meetup to elegant corporate fundraisers.

Location and Hours
Seekonk, Massachusetts;
Monday through Thursday 11am to 9pm; Friday and Saturday, 11am to
midnight
```

Select Save (or "Save as") from the File menu to get the "Save as" dialog box (FIGURE 4-4). The first thing you need to do is create a new folder (click the New Folder button on both Windows and macOS) that will contain all of the files for the site. The technical name for the folder that contains everything is the **local root directory**.

Windows 11



MacOS

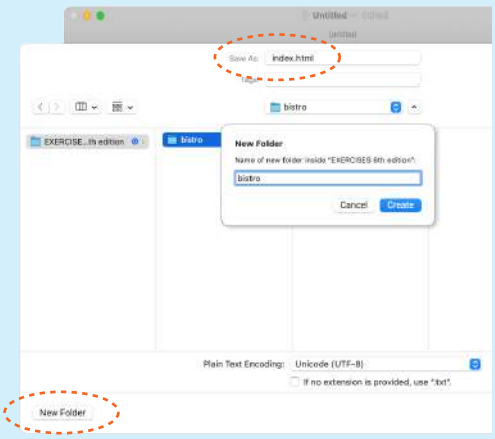


FIGURE 4-4. Saving *index.html* in a new folder called *bistro*

Name the new folder *bistro*, and save the text file in it as *index.html*. The filename needs to end in *.html* to be recognized by the browser as a web document. See the sidebar **"Naming Conventions"** for more tips on naming files.

2. Just for kicks, let's take a look at *index.html* in a browser.

   Windows users*:* Double-click the filename in the File Explorer to launch your default browser, or right-click the file for the option to open it in the browser of your choice.

   Mac users: Launch your favorite browser (I'm using Google Chrome) and choose Open or Open File from the File menu. Navigate to *index.html*, and then select the document to open it in the browser.

3. You should see something like the page shown in FIGURE 4-5. I'll talk about the results in the following section.

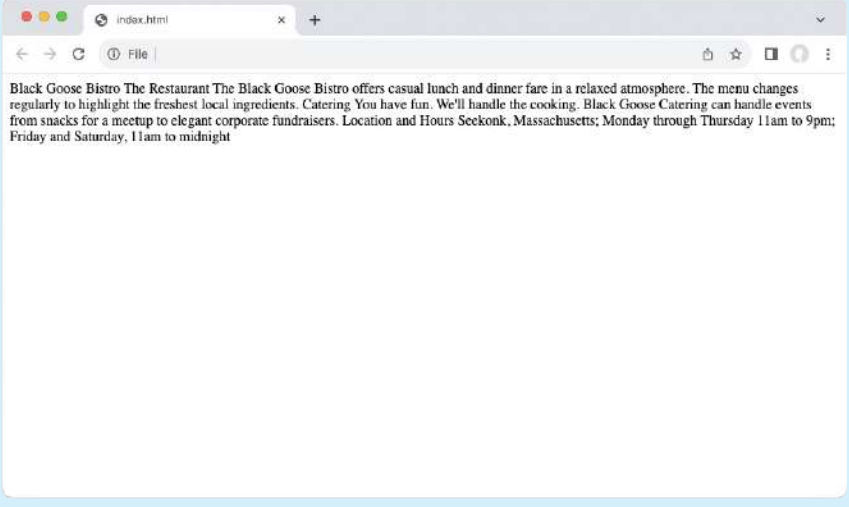

index.html
File
Black Goose Bistro The Restaurant The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients. Catering You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers. Location and Hours Seekonk, Massachusetts; Monday through Thursday 11am to 9pm; Friday and Saturday, 11am to midnight

FIGURE 4-5. A first look at the content in a browser

Naming Conventions

It's important that you follow these rules and conventions when naming your files:

- **Use proper suffixes for your files.** HTML files must end with *.html*. Web images must be labeled according to their file format: *.gif*, *.png*, *.jpg* (*.jpeg* is also acceptable, although less common), or *.svg*.
- **Never use character spaces within filenames.** It is common to use an underline character or hyphen to visually separate words within filenames, such as *robbins_bio.html* or *robbins-bio.html*.
- **Avoid special characters** such as *?*, *%*, *#*, */*, *:*, *;*, *•*, etc. Limit filenames to letters, numbers, underscores, hyphens, and periods (there may not be multiple periods in a row). Also avoid international characters, such as the Swedish å.
- **Filenames may be case-sensitive,** depending on your server configuration. Consistently using all lowercase letters in filenames, although not required, is one way to make your filenames easier to manage.
- **Keep filenames short.** Long names are more likely to be misspelled, and short names shave a few extra bytes off the file size. If you really must give the file a long, multiword name, you can separate words with hyphens, as in *a-long-document-title.html*, to improve readability.
- **Self-imposed conventions.** It's helpful to develop a consistent naming scheme for large sites—for instance, always using lowercase with hyphens between words. This takes some of the guesswork out of remembering what you named a file when you go to link to it later.

Learning from Step 1

Our page isn't looking so good (FIGURE 4-5). The text is all run together into one block—that's not how it looked when we typed it into the original document! There are a couple of lessons to be learned here. The first thing that is apparent is that the browser ignores line breaks in the source document. The sidebar **"What Browsers Ignore"** lists other types of information in the source document that are not displayed in the browser window.

Second, we see that simply typing in some content and naming the document with an *.html* extension is not enough. While the browser can display the text from the file, we haven't indicated the *structure* of the content. That's where HTML comes in. We'll use markup to add structure: first to the HTML document itself (coming up in step 2), then to the page's content (step 3). Once the browser knows the structure of the content, it can display the page in a more meaningful way.

STEP 2: GIVE THE HTML DOCUMENT STRUCTURE

We have our content saved in an HTML document—now we're ready to start marking it up.

The Anatomy of an HTML Element

Back in **Chapter 2,** you saw examples of elements with an opening tag (** ** for a paragraph, for example) and a closing tag (** **). Before we start adding tags to our document, let's look at the anatomy of an HTML element (its **syntax**) and firm up some important terminology. A generic container element is labeled in FIGURE 4-6.

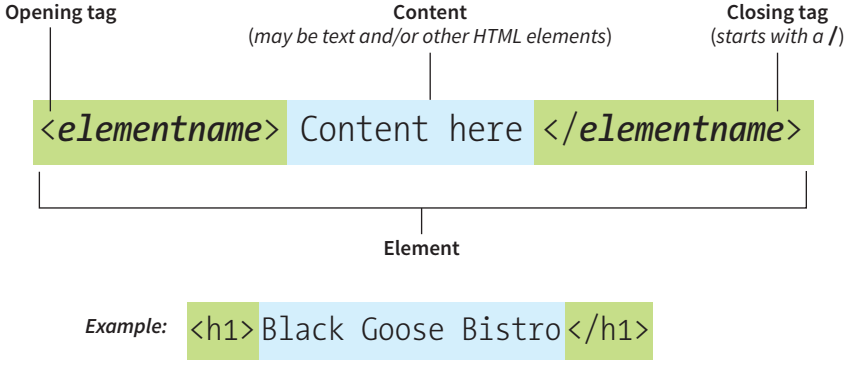


FIGURE 4-6. The parts of an HTML container element

What Browsers Ignore

The following information in the source document will be ignored when it is viewed in a browser:

Multiple character (white) spaces

When a browser encounters more than one consecutive blank character space, it displays a single space. So, if the document contains

```
long,      long      ago
```

the browser displays:

```
long, long ago
```

Line breaks (carriage returns)

Browsers convert carriage returns to spaces, so following the earlier "ignore multiple character spaces" rule, line breaks have no effect on formatting the page.

Tabs

Tabs are also converted to spaces, so guess what? They're useless for indenting text on a web page (although they may make your code more readable).

Unrecognized markup

Browsers are instructed to ignore any tag they don't understand or that was specified incorrectly. Depending on the element and the browser, this can have varied results. The browser may display nothing at all, or it may display the contents of the tag as though it were normal text.

Text in comments

Browsers do not display text between the special **<!--** and **-->** tags used to denote a comment. See the **"Adding Hidden Comments"** sidebar in the discussion of step 3 for more on using comments.

■ MARKUP TIP

Slash Versus Backslash

HTML tags and URLs use the slash character (/). The slash character is found under the question mark (?) on the English QWERTY keyboard (key placement on keyboards in other countries may vary).

It's easy to confuse the slash with the backslash character (\), which is found under the bar character (|); see FIGURE 4-7. The backslash key will not work in tags or URLs, so be careful not to use it.

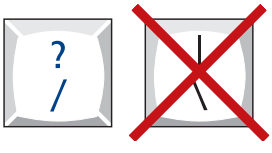


FIGURE 4-7. Slash versus backslash keys

NOTE

There is a stricter version of HTML called XHTML that requires all element and attribute names to appear in lowercase. XHTML is all but obsolete except for certain use cases when it is combined with other XML languages, but the preference for all-lowercase element names has persisted. Other strict requirements of XHTML, such as always including closing tags and always putting attribute values in quotation marks, also persist as good HTML form even though they are not required. You can learn more about XHTML in the article "HTML+ to HTML5," available at learningwebdesign.com/articles.

Elements are identified by tags in the text source. A **tag** consists of the element name (usually an abbreviation of a longer descriptive name) within angle brackets (`< >`). The browser knows that any text within brackets is hidden and not displayed in the browser window.

The element name appears in the **opening tag** (also called a **start tag**) and again in the **closing** (or **end**) **tag**, where it is preceded by a slash (**/**). The closing tag works something like an "off" switch for the element. Be careful not to use the similar backslash character in end tags (see the tip **"Slash Versus Backslash"**).

The tags added around content are referred to as the **markup**. It is important to note that an **element** consists of both the content *and* its markup (the start and end tags). Not all elements have content, however. Some are void by definition, such as the **`img`** element used to add an image to the page. I'll talk about void elements a little later in this chapter.

One last thing: capitalization. In HTML, the capitalization of element names is not important. That is, it is not case-sensitive (unlike the stricter XHTML; see **Note**). So, **`<img>`**, **`<Img>`**, and **`<IMG>`** are all the same as far as the browser is concerned. However, most developers prefer the consistency of writing element names in all lowercase, as I will be doing throughout this book.

Basic Document Setup

FIGURE 4-8 shows the recommended minimal skeleton of an HTML document. I say "recommended" because the only element that is *required* in HTML is the **`title`**. But I feel it is better, particularly for beginners, to explicitly organize documents into metadata (**`head`**) and content (**`body`**) areas.

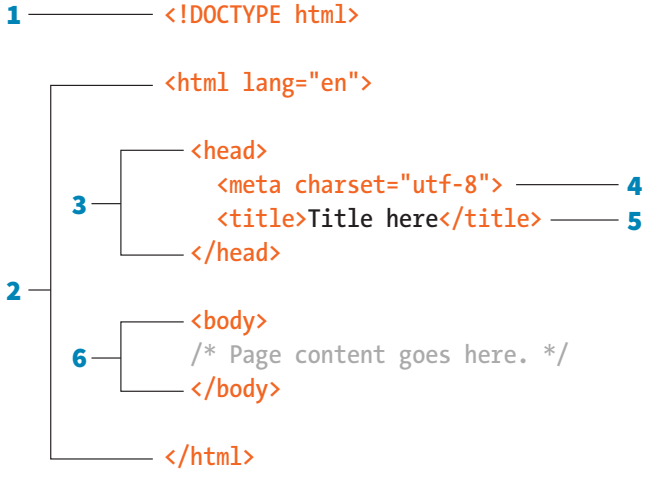


FIGURE 4-8. The minimal structure of an HTML document includes **`head`** and **`body`** elements contained within the **`html`** root element.

Let's take a look at what's going on in this minimal markup example:

1. Not to confuse things, but the first line in the example isn't an element at all. It's a **document type declaration** (also called a **DOCTYPE declaration**) [**1**], which is required to let modern browsers know that the document is written in compliance with the current HTML specification (see **Note**).

NOTE

In the past, the W3C maintained many versions of HTML concurrently, each requiring a different level of strictness in how the document was authored (see the article "HTML+ to HTML5" at learningwebdesign.com/articles for details). In those days, the DOCTYPE let the browser know which of those specifications to use. Thankfully, it's more straightforward now.

2. The entire document is contained within an `html` element [**2**]. The `html` element is called the **root element** because it contains all the elements in the document. The `html` element may not be contained within any other element. The `lang` attribute specifies the language in which the document is written. It is not required by the spec, but including it is a best practice that aids accessibility. In the example, `en` is a **language tag** that indicates our content is written in English, which is also the default if the `lang` attribute is omitted. The full list of language abbreviations is maintained by the Internet Assigned Numbers Authority (IANA) at *oreil.ly/99esV*.

3. Within the `html` element, the document is divided into a **head** and a **body**. The `head` element [**3**] contains elements that pertain to the document that are not rendered as part of the content, such as its title, style sheets, scripts, and metadata.

4. `meta` elements provide document **metadata**, which is information about the document [**4**]. In this case, the `meta` element specifies the **character encoding** (a standardized collection of letters, numbers, and symbols) used in the document as Unicode version UTF-8 (see the sidebar **"Introducing Unicode"**). I don't want to go into too much detail on this right now, but know that there are many good reasons for specifying the `charset` in every document, so I have included it as part of the minimal document markup. Other types of metadata that can be provided by the `meta` element are the author, keywords, publishing status, and a description that can be used by search engines.

5. Also in the `head` is the mandatory `title` element [**5**]. According to the HTML specification, every document must contain a descriptive title.

6. Finally, the `body` element [**6**] contains everything that we want to show up in the browser window.

Are you ready to start marking up the Black Goose Bistro home page? Open the *index.html* document in your text editor and move on to EXERCISE 4-2.

Introducing Unicode

All the characters that make up languages are stored in computers as numbers. A standardized collection of characters with their reference numbers (**code points**) is called a **coded character set**, and the way in which those characters are converted to bytes for use by computers is the **character encoding**. In the early days of computing, computers used limited character sets such as ASCII that contained 128 characters (letters from Latin languages, numbers, and common symbols). The early web used the Latin-1 (ISO 8859-1) character encoding that included 256 Latin characters from most Western languages—but given that the web was "worldwide," this was clearly not sufficient.

Enter Unicode. **Unicode** (also called the **Universal Character Set**) is a super-character set that contains over 150,000 characters (letters, numbers, symbols, ideograms, logograms, etc.) from all active modern languages. You can read all about it at *Unicode.org*. Unicode has three standard encodings—UTF-8, UTF-16, and UTF-32—that differ in the number of bytes used to represent the characters (1, 2, or 4, respectively).

HTML uses the UTF-8 encoding by default, which allows wide-ranging languages to be mixed within a single document. It is always a good idea to declare the character encoding for a document with the `meta` element, as shown in the previous example. Your server also needs to be configured to identify HTML documents as UTF-8 in the **HTTP header** (information about the document that the server sends to the user agent). You can ask your server administrator to confirm the encoding of the HTML documents.

EXERCISE 4-2. Adding minimal markup

1. Open the new *index.html* document if it isn't open already, and add the DOCTYPE declaration:

```
<!DOCTYPE html>
```

2. Put the entire document in an HTML root element by adding an **<html>** start tag after the DOCTYPE and an **</html>** end tag at the very end of the text. Add the **lang** attribute to indicate the document is written in English.

3. Next, create the document head that contains the title for the page. Insert **<head>** and **</head>** tags before the content. Within the **head** element, add information about the character encoding, **<meta charset="utf-8">**, and the title, "Black Goose Bistro", surrounded by opening and closing **<title>** tags.

4. Finally, define the body of the document by wrapping the text content in **<body>** and **</body>** tags. When you are done, the source document should look like this (the markup is shown in color to make it stand out):

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Black Goose Bistro</title>
</head>

<body>
Black Goose Bistro

...(remaining content)...

</body>
</html>
```

5. Save the document in the *bistro* directory, so that it overwrites the old version. Open the file in the browser, or hit Refresh or Reload if it is open already. FIGURE 4-9 shows how it should look now.

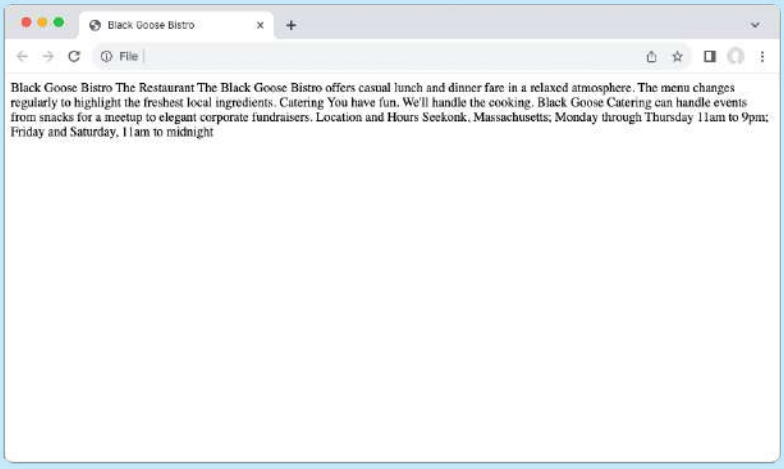

Black Goose Bistro
File
Black Goose Bistro The Restaurant The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients. Catering You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers. Location and Hours Seekonk, Massachusetts; Monday through Thursday 11am to 9pm; Friday and Saturday, 11am to midnight

FIGURE 4-9. The page in a browser after the document structure elements have been defined.

Not much has changed in the Bistro page after setting up the document, except that the browser now displays the title of the document in the top bar or tab (FIGURE 4-9). If someone were to bookmark this page, that title would be added to their Bookmarks or Favorites list as well (see the sidebar **"Don't Forget a Good Title"**). But the content still runs together, because we haven't given the browser any indication of how it should be structured. We'll take care of that next.

STEP 3: IDENTIFY ELEMENTS

With a little markup experience under your belt, it should be easy to add the markup for headings and subheads (`h1` and `h2`), paragraphs (`p`), and emphasized text (`em`) to our content, as we'll do in EXERCISE 4-3. However, before we begin, let's take a moment to talk about what we're doing and not doing when marking up content with HTML.

Mark It Up Semantically

The purpose of HTML is to add meaning and structure to the content. It is *not* intended to describe how the content should look (see **Note**).

Your job when marking up content is to choose the HTML element that provides the most meaningful description of the content at hand. In the biz, we call this **semantic markup**. For example, the most important heading at the beginning of the document should be marked up as an `h1` because it is the most important heading on the page. Don't worry about what it looks like; you can easily change that with a style sheet. The important thing is that you choose elements based on what makes the most sense for the content.

It is worth noting that you can't make up your own markup elements; you need to choose from the elements defined in the HTML standard (see **Note**).

In addition to adding meaning to content, the markup creates the document's structure. The way elements follow each other or nest within one another creates relationships between them. You can think of this structure as an outline (its technical name is the **DOM**, for **Document Object Model**).

There are a lot of benefits to having a clear, well-defined document hierarchy. The underlying hierarchy gives browsers, and accessibility technology like screen readers, cues on how to handle the content. The document hierarchy is also the foundation upon which we add presentation instructions with style sheets and behaviors with JavaScript.

OK, enough lecturing. It's time to get to work on that content in EXERCISE 4-3.

Don't Forget a Good Title

A `title` element is not only required for every document but also quite useful. The title is what displays in a user's Bookmarks or Favorites list and on tabs in desktop browsers. Descriptive titles are also a key tool for improving accessibility, as they are the first things a person hears when using a screen reader. Search engines rely heavily on document titles as well.

For these reasons, it's important to provide thoughtful and descriptive titles for all your documents and avoid vague titles, such as "Welcome" or "My Page." You may also want to keep the length of your titles in check so they are able to display in the browser's title area. Knowing that users typically have a number of tabs open or a long list of Bookmarks, put your most uniquely identifying information in the first 20 or so characters.

NOTE

In the '90s, before we had style sheets, HTML provided ways to change the appearance of fonts, colors, and alignment using markup alone. Those presentational extras (like the `font` *element) were kicked to the curb long ago and are not in the current spec. You you may run across them if you view the source of an archived site or in the results of a Google search.*

NOTE

Technically, there is a way to create your own custom elements and define their behaviors by using JavaScript as part of the Web Components specification. See **Chapter 25, Next-Level JavaScript** *for an introduction.*

EXERCISE 4-3. Defining text elements

1. Open the document *index.html* in your text editor, if it isn't open already.
2. The first line of text, "Black Goose Bistro," is the main heading for the page, so we'll mark it up as a heading level 1 (**h1**) element. Put the opening tag, **<h1>**, at the beginning of the line and the closing tag, **</h1>**, after it, like this:

   ```
   <h1>Black Goose Bistro</h1>
   ```

3. Our page also has three subheads. Mark them up as heading level 2 (**h2**) elements in a similar manner. I'll do the first one here; you do the same for "Catering" and "Location and Hours":

   ```
   <h2>The Restaurant</h2>
   ```

4. Each **h2** element is followed by a brief paragraph of text, so let's mark those up as paragraph (**p**) elements in a similar manner. Here's the first one; you do the rest:

   ```
   <p>The Black Goose Bistro offers casual lunch and dinner fare in a
   relaxed atmosphere. The menu changes regularly to highlight the freshest
   local ingredients.</p>
   ```

5. Finally, in the Catering section, I want to emphasize that visitors should just leave the cooking to us. To turn that sentence into "emphasized text," mark it up in an emphasis (**em**) element, as shown here:

   ```
   <p>You have fun. <em>We'll handle the cooking.</em> Black Goose Catering
   can handle events from snacks for a meetup to elegant corporate
   fundraisers.</p>
   ```

6. Now that we've marked up the document, let's save it as we did before, and open (or reload) the page in the browser. You should see a page that looks much like the one in FIGURE 4-10. If it doesn't, check your markup to be sure that you aren't missing any angle brackets or a slash in a closing tag.

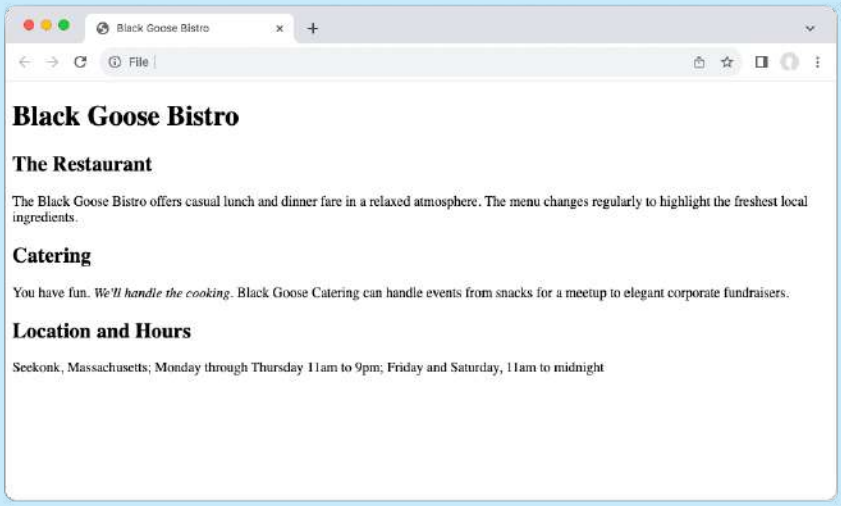

Black Goose Bistro
File
Black Goose Bistro
The Restaurant
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.
Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.
Location and Hours
Seekonk, Massachusetts; Monday through Thursday 11am to 9pm; Friday and Saturday, 11am to midnight

FIGURE 4-10. The home page after the content has been marked up with HTML elements

Now we're getting somewhere. With the elements properly identified, the browser can now display the text in a more meaningful manner. There are a few significant things to note about what's happening in FIGURE 4-10; we'll look at those in the following sections.

Block and Text-Level Elements

Although it may seem like stating the obvious, it's worth pointing out that the heading and paragraph elements in our example start on new lines and do not run together as they did before. That's because by default, headings and paragraphs display as **block elements**. Browsers treat block elements as though they are in little rectangular boxes, stacked up in the page. Each block element begins on a new line, and some space is also usually added above and below the entire element by default. In FIGURE 4-11, the edges of the block elements are outlined in red.

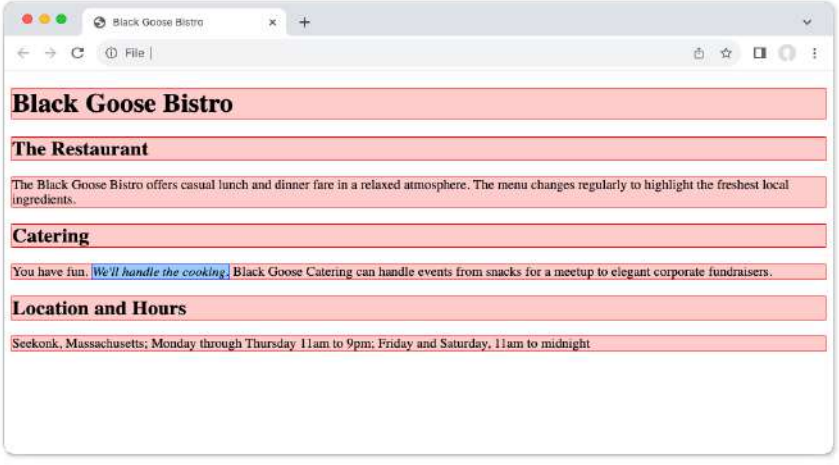

Black Goose Bistro
File |
Black Goose Bistro
The Restaurant
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.
Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.
Location and Hours
Seekonk, Massachusetts; Monday through Thursday 11am to 9pm; Friday and Saturday, 11am to midnight

FIGURE 4-11. The outlines show the structure of the elements in the home page.

By contrast, look at the text we marked up as emphasized (`em`, outlined in blue in FIGURE 4-11). It does not start a new line, but rather stays in the flow of the paragraph. That's because the `em` element is a **text-level element** (also called a **phrasing element** or **inline element**). By default, text-level elements do not start new lines; they just go with the flow.

Default Styles

The other thing that you will notice about the marked-up page in FIGURES 4-10 and 4-11 is that the browser makes an attempt to give the page some visual hierarchy by making the first-level heading the biggest and boldest thing on the page, with the second-level headings slightly smaller, and so on.

Adding Hidden Comments

You can leave notes in the source document for yourself and others by marking them up as **comments**. Anything you put between comment tags (`<!--` and `-->`) will not display in the browser and will not have any effect on the rest of the source:

```
<!-- This is a comment -->
<!-- This is a
  multiple-line comment
  that ends here. -->
```

Comments are useful for labeling and organizing long documents, particularly when they are shared by a team of developers. In this example, comments are used to point out the section of the source document that contains the navigation:

```
<!-- start global nav -->
<ul>
 ...
</ul>
<!-- end global nav -->
```

Bear in mind that although the browser will not display comments in the web page, readers can see them if they "view source," so be sure that the comments you leave are appropriate for everyone.

How does the browser determine what an `h1` should look like? It uses a style sheet! All browsers have their own built-in style sheets (called **user agent style sheets** in the HTML spec) that describe the default rendering of elements. The default rendering is similar from browser to browser (for example, `h1`s are always big and bold), but there are some variations (the `blockquote` element for long quotes may or may not be indented, for instance).

If you think the `h1` is too big and clunky as the browser renders it, change it with your own style sheet rule. Resist the urge to mark up the heading with another element just to get it to look better—for example, using an `h3` instead of an `h1` so it isn't as large. You should always choose elements based on how accurately they describe the content, and don't worry about the browser's default rendering.

We'll fix the presentation of the page with style sheets in a moment, but first, let's add an image to the page.

STEP 4: ADD AN IMAGE

What fun is a web page with no images? In EXERCISE 4-4, we'll add an image to the page with the `img` element. Images will be discussed in more detail in **Chapter 7, Adding Images**, but for now, they give me an opportunity to introduce two more basic markup concepts: void elements and attributes.

Void Elements

So far, nearly all of the elements we've used in the Black Goose Bistro home page have followed the syntax shown in FIGURE 4-6: a bit of text content surrounded by start and end tags.

A handful of elements, however, do not have content because they are used to provide simple directives. These elements are said to be **void** (previously called **empty**, so you may also hear that term). The image element (`img`) is an example of a void element. It tells the browser to get an image file from the server and embed it at that spot in the flow of the text. Other void elements include the line break (`br`), thematic breaks (`hr`, a.k.a. "horizontal rules"), and elements that provide information about a document but don't affect its displayed content, such as the `meta` element that we used earlier (see **Note**).

NOTE

The complete list of void elements as defined in the spec is `area`, `base`, `br`, `col`, `embed`, `hr`, `img`, `input`, `link`, `meta`, `source`, `track`, *and* `wbr`.

FIGURE 4-12 shows the very simple syntax of a void element (compare it to FIGURE 4-6).

```
<element-name>
```

Example: The **br** element inserts a line break.

```
<p>1005 Gravenstein Highway North<br>Sebastopol, CA 95472</p>
```

FIGURE 4-12. Void (empty) element structure

Attributes

Let's get back to adding an image with the **img** element. Obviously, an **<img>** tag is not very useful by itself—it doesn't indicate which image to use. That's where attributes come in. **Attributes** are instructions that clarify or modify an element. For the **img** element, the **src** (short for "source") attribute is required, and specifies the location (URL) of the image file. The **alt** attribute is also required for providing a description of the image (we'll talk more about alternative text in **Chapter 7**).

The syntax for an attribute is as follows:

```
attributeName="value"
```

Attributes go after the element name, separated by a space. In content (non-void) elements, attributes go in the opening tag only:

```
<element attributeName="value">
<element attributeName="value">Content</element>
```

You can also put more than one attribute in an element, in any order. Just keep them separated with spaces:

```
<element attribute1="value1" attribute2="value2">
```

FIGURE 4-13 shows an **img** element with its required attributes labeled.

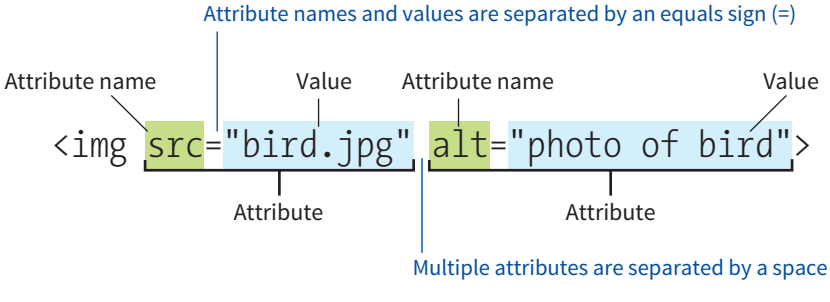


FIGURE 4-13. An **img** element with two attributes

What Is That Extra Slash?

If you poke around in source documents for existing web pages, you may see void elements with extra slashes at the end, like so: **<img />**, **
, **<meta />, and **<hr />**. That indicates the document was written according to the stricter rules of XHTML. In XHTML, *all* elements, including void elements, must be closed (or **terminated**, to use the proper term). You terminate void elements by adding a trailing slash before the closing bracket. The preceding character space is not required but was used for backward compatibility with browsers that did not have XHTML parsers, so **<img/>**, **
**, and so on are also valid. It is no longer required to self-close void elements, so I will not be doing so in this book. Some developers continue to do so out of habit or preference.

Attributes are instructions that clarify or modify an element.

Here's what you need to know about attributes:

- Attributes go after the element name in the opening tag only, never in the closing tag.
- There may be several attributes applied to an element, separated by spaces in the opening tag. Their order is not important.
- Most attributes take values, which follow an equals sign (=). In HTML, some attribute values are single descriptive words. For example, the `checked` attribute, which makes a form checkbox checked when the form loads, is equivalent to `checked="checked"`. You may hear this type of attribute called a **Boolean attribute** because it describes a feature that is either on or off.
- A value might be a number, a word, a string of text, a URL, or a measurement, depending on the purpose of the attribute.
- Wrapping attribute values in double quotation marks is a strong convention, but note that quotation marks are not required and may be omitted. In addition, either single or double quotation marks are acceptable, as long as the opening and closing marks match. Note that quotation marks in HTML files need to be straight (") , not curly (”).
- The attribute names and values available for each element are defined in the HTML specification; in other words, you can't make up an attribute for an element.
- Some attributes are required, such as the `src` and `alt` attributes in the `img` element. The HTML specification also defines which attributes are required in order for the document to be valid.

Now you should be more than ready to try your hand at adding the `img` element with its attributes to the Black Goose Bistro page in EXERCISE 4-4. We'll throw a few line breaks in there as well.

EXERCISE 4-4. Adding an image

1. If you're working along, the first thing you'll need to do is get a copy of the image file on your hard drive so you can see it in place when you open the file locally. The image file is provided with the materials for this chapter. You can also get the image file by saving it right from the sample web page online, at *learningwebdesign.com/6e/materials/bistro*. Right-click (or Control-click on a Mac) the goose image and select "Save image" (or similar) from the pop-up menu, as shown in FIGURE 4-14. Name the file *blackgoose.png*. Be sure to save it in the *bistro* folder with *index.html*.
2. Once you have the image, insert it before the first-level heading by typing in the `img` element and its attributes as shown here:

```
<img src="blackgoose.png" alt="black goose logo">
<h1>Black Goose Bistro</h1>
```

Windows: Right-click on the image to access the pop-up menu.

Mac (shown): Control-click on the image to access the pop-up menu. The options may vary by browser.

FIGURE 4-14. Saving an image file from a page on the web

The `src` attribute provides the location and name of the image file that should be inserted, and the `alt` attribute provides text that should be displayed if the image is not available. Both of these attributes are required in every `img` element in order for the document to be valid.

3. Make it clear that the image and heading work together as the header of the document by wrapping the elements in opening and closing `<header>` tags. This also allows us to apply styles to the elements as a unit in the next exercise:

```
<header>
  <img src="blackgoose.png" alt="black goose logo">
  <h1>Black Goose Bistro</h1>
</header>
```

4. Let's break up the last paragraph into three lines for better clarity. Drop line break (`<br>`) tags at the spots where you'd like the line breaks to occur. Try to match the screenshot in FIGURE 4-15.

5. Now save *index.html* and open or refresh it in the browser window. The page should look like the one shown in FIGURE 4-15. If it doesn't, check to make sure that the image file, *blackgoose.png*, is in the same directory as *index.html*. If it is, then check to make sure that you aren't missing any characters, such as a closing quote or bracket, in the `img` element markup.

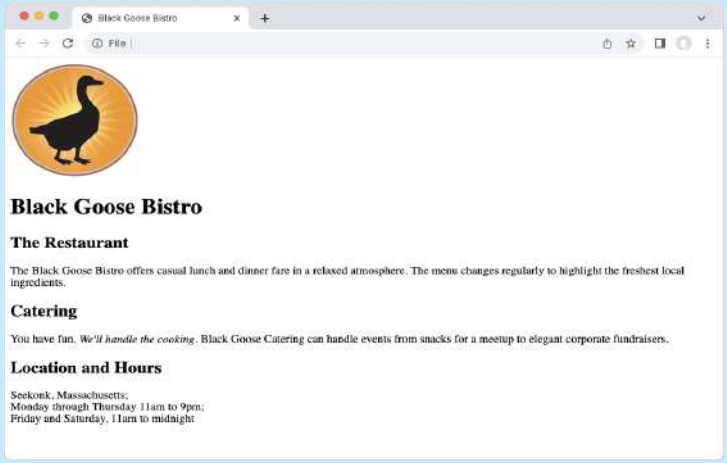

Black Goose Bistro
Black Goose Bistro
The Restaurant
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.
Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.
Location and Hours
Seekonk, Massachusetts;
Monday through Thursday 11am to 9pm;
Friday and Saturday, 11am to midnight

FIGURE 4-15. The Black Goose Bistro page with the logo image.

STEP 5: CHANGE THE LOOK WITH A STYLE SHEET

Depending on the content and purpose of your website, you may decide that the browser's default rendering of your document is perfectly adequate. However, we're going to pretty up the Black Goose Bistro home page a bit to make a good first impression on potential patrons. "Prettying up" is just my way of saying that I'd like to change its presentation, which is the job of Cascading Style Sheets (CSS).

In EXERCISE 4-5, we'll change the appearance of the page by using some simple style sheet rules. Don't worry about understanding them all right now. I'll get into CSS in more detail in **Part III, CSS for Presentation**. But I want to at least give you a taste of what it means to add a "layer" of presentation onto the structure we've created with our markup.

EXERCISE 4-5. Adding a style sheet

1. Open *index.html*. We're going to use the **style** element to apply a very simple embedded style sheet to the page. This is just one of the ways to add a style sheet; the others are covered in **Chapter 11, Introducing Cascading Style Sheets**.
2. The **style** element is placed inside the document **head**. Start by adding the **style** element to the document as shown here:

```
<head>
  <meta charset="utf-8">
  <title>Black Goose Bistro</title>
  <style>
  </style>
</head>
```

3. Next, type the following style rules within the **style** element just as you see them here. Don't worry if you don't know exactly what's going on (although it's fairly intuitive); you'll learn all about style rules in **Part III**. You can also copy and paste these style rules from the finished document in the *materials* folder.

```
<style>
  body {
    margin: 0 10%;
    font: 1.25rem/1.75rem sans-serif;
    color: floralwhite;
    background-color: darkslategray;
    }
  header {
    background-color: #faf2e4;
    text-align: center;
    padding: 12px;
    margin-bottom: 36px;
    }
  h1 {
    color: darkslategray;
    font: bold 1.75rem sans-serif;
    text-transform: uppercase;
    letter-spacing: 5px;
    margin: 0;
    }
  h2 {
    color: orange;
    font: bold 1.25rem sans-serif;
    text-align: center;
    }
</style>
```

4. Now it's time to save the file and take a look at it in the browser. It should look like the page in FIGURE 4-16. If it doesn't, go over the style sheet to make sure you didn't miss a semicolon or a curly brace. Compare the way the page looks with our styles to how it looked with the browser's default styles (FIGURE 4-15).

BLACK GOOSE BISTRO
The Restaurant
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.
Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.
Location and Hours
Seekonk, Massachusetts;
Monday through Thursday 11am to 9pm;
Friday and Saturday, 11am to midnight

FIGURE 4-16. The Black Goose Bistro page after CSS style rules have been applied

We're finished with the Black Goose Bistro page. Not only have you written your first web page, complete with a style sheet, but you've also learned about elements, attributes, void elements, block and inline elements, the basic structure of an HTML document, and the correct use of markup along the way. Not bad for one chapter!

WHEN GOOD PAGES GO BAD

The previous demonstration went smoothly, but it's easy for small things to go wrong when you're typing out HTML markup by hand. Unfortunately, one missed character can break a whole page. Let's try breaking some elements on purpose so you can see what happens.

What if we had neglected to type the slash in the closing emphasis tag (`</em>`) in EXERCISE 4-3? With just one character out of place, the remainder of the document displays in emphasized (italic) text, as shown in FIGURE 4-17. That's because without that slash, there's nothing telling the browser to "turn off" the emphasized formatting, so it just keeps going (see **Note**).

```
<h2>Catering</h2>
<p>You have fun. <em>We'll handle the cooking.<em> Black Goose
Catering can handle events from snacks for a meetup to elegant
corporate fundraisers.</p>
```

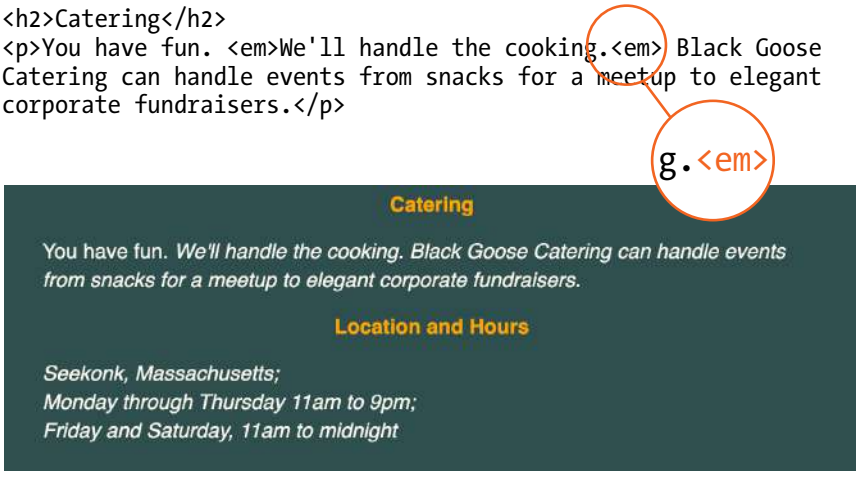

g.
Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.
Location and Hours
Seekonk, Massachusetts;
Monday through Thursday 11am to 9pm;
Friday and Saturday, 11am to midnight

FIGURE 4-17. When a slash is omitted, the browser doesn't know when the element ends, as is the case in this example.

We've fixed the slash, but this time, let's see what would have happened if we had accidentally omitted a bracket from the end of the first `<h2>` tag (FIGURE 4-18).

See how the headline is missing? That's because without the closing tag bracket (`>`), the browser assumes that all the following text—all the way up to the next closing bracket it finds—is part of the `<h2>` opening tag. Browsers don't display any text within a tag, so the heading disappeared. The browser just ignored the long, foreign-looking element name and moved on to the next element.

NOTE

Omitting the slash in the closing tag (or even omitting the closing tag itself) for block elements, such as headings or paragraphs, may not be so dramatic. Browsers interpret the start of a new block element to mean that the previous block element is finished.

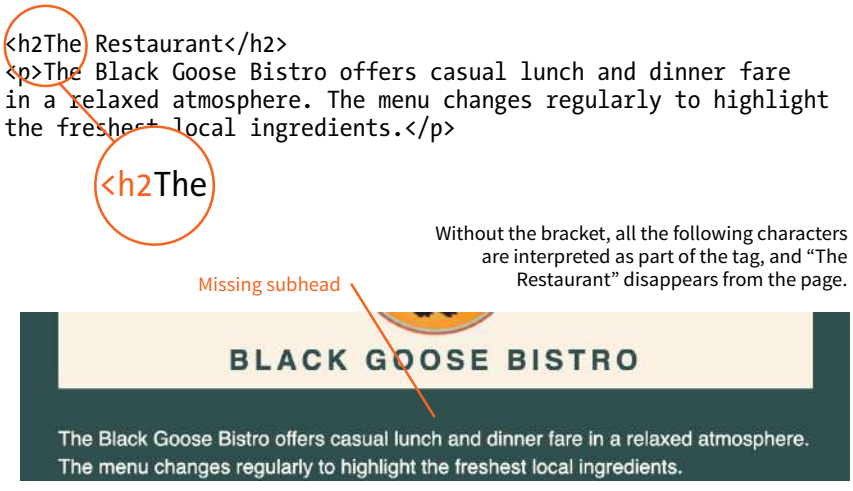

<h2The Restaurant</h2>
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.
<h2The
Without the bracket, all the following characters are interpreted as part of the tag, and "The Restaurant" disappears from the page.
Missing subhead
BLACK GOOSE BISTRO
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere.
The menu changes regularly to highlight the freshest local ingredients.

FIGURE 4-18. A missing end bracket makes the browser think the following characters are part of the tag, and therefore the headline text doesn't display.

Making mistakes in your first HTML documents and fixing them is a great way to learn. If you write your first pages perfectly, try fiddling with the code to see how the browser reacts to various changes. This can be extremely useful in troubleshooting pages later. The sidebar **"Having Problems?"** lists some common problems. Note that these issues are not specific to beginners. Little stuff like this goes wrong all the time, even for the pros.

Having Problems?

The following are some typical problems that crop up when you are creating web pages and viewing them in a browser:

I've changed my document, but when I reload the page in my browser, it looks exactly the same.

It could be you didn't save your document before reloading, or you may have saved it in a different directory.

Half my page disappeared.

This could happen if you are missing a closing bracket (>) or a quotation mark within a tag. This is a common error when you're writing HTML by hand.

I put in an image with the `img` *element, but all that shows up is a broken image icon.*

The broken icon could mean a couple of things. First, it might mean that the browser is not finding the image file. Make sure that the URL to the file is correct (we'll discuss URLs further in **Chapter 6, Adding Links**), and that the image file is actually in the directory you've specified. If the file is there, make sure it is in one of the formats that web browsers can display (such as PNG, JPEG, WebP, GIF, or SVG) and that it is named with the proper suffix (*.png*, *.jpg*, *.jpeg*, *webp*, *.gif*, or *.svg*, respectively).

VALIDATING YOUR DOCUMENTS

One way that professional web developers catch errors in their markup is by validating their documents. What does that mean? To **validate** a document is to check your markup to make sure that you have abided by all the rules of HTML. Documents that are error-free are said to be **valid**. It is strongly recommended that you validate your documents, especially for professional sites. Valid documents are more consistent on a variety of browsers, display more quickly, and are more accessible.

Browsers are very forgiving and don't require documents to be valid (in other words, they'll do their best to display them, errors and all), but anytime you stray from the standard, you introduce unpredictability in the way the page is handled by browsers or alternative devices.

So how do you make sure your document is valid? You could check it yourself or ask a friend, but humans make mistakes, and you aren't expected to memorize every minute rule in the specifications. Instead, use a **validator**, software that checks your source against the HTML version you specify. These are some things validators check for:

- The inclusion of a DOCTYPE declaration
- The inclusion of required rules and attributes
- Nonstandard elements
- Mismatched tags
- Nesting errors (incorrectly putting elements inside other elements)
- Typos and other minor errors

Developers use a variety of helpful tools for checking and correcting errors in HTML documents. Many code editors, frameworks, and other developer build tools have validators built in. Browsers such as Safari and Chrome feature a number of developer tools, including validators, so you can check your work on the fly. Also, a lot of code editors nest and close elements and attributes automatically as you type, so there is less room for error in the writing process itself.

The Nu Html Checker (***validator.nu***) is a popular online validator. You can upload a file, or provide a link to a page that is already online. FIGURE 4-19 shows the report the validator generates when I upload the version of the Bistro *index.html* file that doesn't have any markup. It shows that there are a number of missing elements that keep this document from being valid. The report also highlights the problem source code and provides an explanation of how the code should appear. Pretty darned handy!

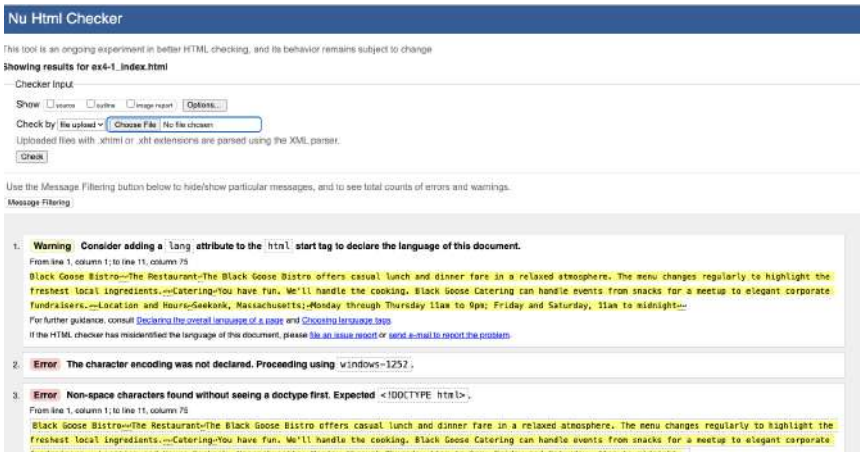

Nu Html Checker
This tool is an ongoing experiment in better HTML checking, and its behavior remains subject to change
Showing results for ex4-1_index.html
Checker Input
Show
Options...
Check by
File upload
Choose File
No file chosen
Uploaded files with .xhtml or .xht extensions are parsed using the XML parser.
Check
Use the Message Filtering button below to hide/show particular messages, and to see total counts of errors and warnings.
Message Filtering
1. Warning Consider adding a lang attribute to the html start tag to declare the language of this document.
From line 1, column 1; to line 11, column 75
Black Goose Bistro—The Restaurant—The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.—Catering—You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.—Location and Hours—Seekonk, Massachusetts;—Monday through Thursday 11am to 9pm; Friday and Saturday, 11am to midnight—
For further guidance, consult Declaring the overall language of a page and Choosing language tags
If the HTML checker has misidentified the language of this document, please file an issue report or send e-mail to report the problem.
2. Error The character encoding was not declared. Proceeding using windows-1252.
3. Error Non-space characters found without seeing a doctype first. Expected <!DOCTYPE html>.
From line 1, column 1; to line 11, column 75
Black Goose Bistro—The Restaurant—The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.—Catering—You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate

FIGURE 4-19. Using the Nu Html Checker to validate an HTML document

HTML RESOURCES

Now you have a little hands-on experience marking up a document and a feel for HTML. In the upcoming chapters, I will present collections of HTML elements organized by topic: text content, links, images, tables, forms, and embedded media. By the end of **Part II**, you will have gotten a thorough introduction to the majority of the elements in the HTML specification—certainly the ones you are most likely to need when you're just starting out (and many you may never use at all). If you feel you need more explanation, I encourage you to get familiar with these sites, which are excellent resources for students as well as experienced developers:

MDN Web Docs *(developer.mozilla.org)*

This site is many developers' go-to for information about HTML and other web technologies. It goes into deeper technical detail than I do here, so it will help you take your skills to the next level and also help when you're stuck. It is published by Mozilla, the creators of the Firefox browser.

Web.dev *(web.dev)*

Google's Chrome Developer Relations team publishes comprehensive courses and articles about HTML, CSS, and JavaScript.

W3Schools *(w3schools.com)*

W3Schools publishes an HTML element reference (*w3schools.com/tags/*) as well as succinct tutorials on HTML and many other topics. Exercises and quizzes are available if you'd like to put your knowledge to the test, and W3Schools also offers courses and certification exams for a fee.

The HTML Living Standard *(html.spec.whatwg.org)*

Get the definitive answer on all matters of HTML in the current spec.

TEST YOURSELF

Now is a good time to make sure you understand the basics of markup. Use what you've learned in this chapter to answer the following questions. Answers are in **Appendix A**.

1. What is the difference between a tag and an element?

2. Write out the recommended minimal markup for an HTML document.

3. Indicate whether each of these filenames is an acceptable name for a web document by circling "Yes" or "No." If it is not acceptable, provide the reason:

a. Sunflower.html	Yes	No
b. index.doc	Yes	No
c. cooking home page.html	Yes	No
d. Song_Lyrics.html	Yes	No
e. games/rubix.html	Yes	No
f. %whatever.html	Yes	No

4. All of the following markup examples are incorrect or invalid. Describe what is wrong with each one, and then write it correctly.

   a. `<img "birthday.jpg">`

   b. `<em>Congratulations!<em>`

   c. `<a href="file.html">linked text</a href="file.html">`

   d. `<p>This is a new paragraph<\p>`

5. How would you mark up this comment in an HTML document so that it doesn't display in the browser window?

   `product list begins here`

ELEMENT REVIEW: HTML DOCUMENT SETUP

This chapter introduced the following elements that establish the metadata and content portions of an HTML document. The remaining elements introduced in the exercises will be treated in more depth in upcoming chapters.

Element	Description
`body`	Identifies the body of the document that holds the content
`head`	Identifies the head of the document that contains information about the document itself
`html`	Is the root element that contains all the other elements
`meta`	Provides information about the document
`title`	Gives the page a title

CHAPTER

5

MARKING UP TEXT

Once your content is ready to go (you've proofread it, right?) and you've added the markup to structure the document (`<!DOCTYPE>`, `html`, `head`, `title`, `meta charset`, and `body`), you're ready to identify the elements in the content. This chapter introduces the elements you have to choose from for marking up text. There probably aren't as many of them as you might think, and really just a handful that you'll use with regularity. That said, this chapter is a big one and covers a lot of ground.

As we begin our tour of elements, I want to reiterate how important it is to choose elements **semantically**—that is, in a way that most accurately describes the content's meaning. If you don't like how it looks, change it with a style sheet. A semantically marked-up document ensures your content is available and accessible in the widest range of browsing environments, from desktop computers and mobile devices to assistive screen readers. It also allows nonhuman readers, such as search engine indexing programs, to correctly parse your content and make decisions about the relative importance of elements on the page.

With these principles in mind, it's time to meet the HTML text elements, starting with the most basic element of them all, the humble paragraph.

IN THIS CHAPTER

Choosing the best element for your content

Paragraphs and headings

Three types of lists

Organizing content into sections

Text-level elements

Generic elements, div and span

Improving Accessibility with ARIA

Special characters

NOTE

This book covers markup according to the HTML Living Standard maintained by the WHATWG (whatwg.org), as published in 2025.

PARAGRAPHS

`<p>...</p>`

Paragraph element

Paragraphs are the most rudimentary elements of a text document. Indicate a paragraph with the `p` element by inserting an opening `<p>` tag at the beginning of the paragraph and a closing `</p>` tag after it, as shown in this example:

```
<p>Serif typefaces have small slabs at the ends of letter strokes.
In general, serif fonts can make large amounts of text easier to
read.</p>

<p>Sans-serif fonts do not have serif slabs; their strokes are square
on the end. Helvetica and Arial are examples of sans-serif fonts. In
general, sans-serif fonts appear sleeker and more modern.</p>
```

Visual browsers nearly always display paragraphs on new lines, with a bit of space between them by default (to use a term from CSS, they are displayed as a **block**). Paragraphs may contain text, images, and text-level elements (also called **phrasing content**), but they may *not* contain headings, lists, sectioning elements, or any elements that typically display as blocks by default.

Technically, it is OK to omit the closing **</p>** tag because it is not required in order for the document to be valid. A browser just assumes it is closed when it encounters the next element with a "block" display type. Most web developers prefer to close paragraphs and all elements for the sake of consistency and clarity. I recommend folks who are just learning markup do the same.

No Naked Text!

It is strongly recommended that all text be enclosed in some sort of element. Text that is not contained within tags is called **naked** or **anonymous** text. Although the document will still be valid, it is a best practice not to leave content bare.

HEADINGS

```
<h1>...</h1>
<h2>...</h2>
<h3>...</h3>
<h4>...</h4>
<h5>...</h5>
<h6>...</h6>
```

Heading elements

In the last chapter, we used the **h1** and **h2** elements to indicate headings for the Black Goose Bistro page. There are actually six levels of headings, from **h1** to **h6**. When you add headings to content, the browser uses them to create a **document outline** for the page. Assistive reading devices such as screen readers use the document outline to help users quickly scan and navigate through a page. Search engines look at heading levels as part of their algorithms, potentially giving information in higher heading levels more weight. For these reasons, it is a best practice to start with the level 1 heading (**h1**) and work down in numerical order.

This example shows the markup for four heading levels:

```
<h1>Type Design</h1>

<h2>Serif Typefaces</h2>
<p>Serif typefaces have small slabs at the ends of letter strokes. In
general, serif fonts can make large amounts of text easier to read.</p>

<h3>Baskerville</h3>

<h4>Description</h4>
<p>Description of the Baskerville typeface.</p>

<h4>History</h4>
<p>The history of the Baskerville typeface.</p>

<h3>Georgia</h3>
<p>Description and history of the Georgia typeface.</p>

<h2>Sans-serif Typefaces</h2>
<p>Sans-serif typefaces do not have slabs at the ends of strokes.</p>
```

The markup in this example would create the following document outline:

1. Type Design
    1. Serif Typefaces
       + text paragraph
        1. Baskerville
            1. Description
               + text paragraph
            2. History
               + text paragraph
        2. Georgia
            + text paragraph
    2. Sans-serif Typefaces
       + text paragraph

By default (that is, using the browser's built-in style sheet), the headings in the example display in bold text, starting in very large type for `h1`s, with each consecutive level in smaller text, as shown in FIGURE 5-1. You can use a style sheet to change their appearance.

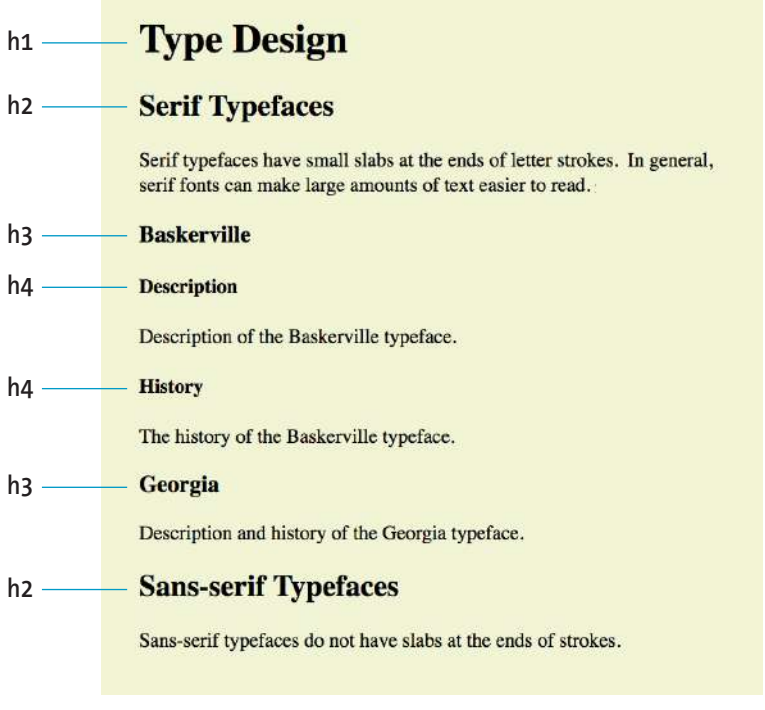


FIGURE 5-1. The default rendering of four heading levels

THEMATIC BREAKS (HORIZONTAL RULES)

`<hr>`
Thematic break (horizontal rule)

If you want to indicate that one topic has completed and another one is beginning, you can insert what the spec calls a "paragraph-level thematic break" with the **hr** element. The **hr** element adds a logical divider between sections of a page or paragraphs, without introducing a new heading level.

In older HTML versions, **hr** was defined as a "horizontal rule" because it inserts a horizontal line on the page. Browsers still render **hr** as a 3D shaded rule and put it on a line by itself with some space above and below by default, but in the current HTML spec, it has a semantic name and definition. If a decorative line is all you're after, it is better to create a rule by specifying a colored border before or after an element with CSS.

hr is a void element, meaning it has no content—you just drop it into place where you want the thematic break to occur, as shown in this example and FIGURE 5-2:

```
<h3>Times</h3>
<p>Description and history of the Times typeface.</p>
<hr>
<h3>Georgia</h3>
<p>Description and history of the Georgia typeface.</p>
```

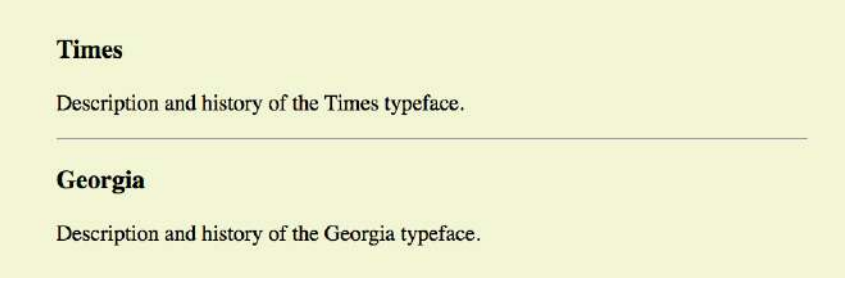


FIGURE 5-2. The default rendering of a thematic break (horizontal rule)

LISTS

Humans are natural list makers, and HTML provides elements for marking up three types of lists:

Unordered lists

Collections of items that appear in no particular order

Ordered lists

Lists in which the sequence of the items is important

Description lists

Lists that consist of name and value pairs, including but not limited to terms and definitions

All list elements—the lists themselves and the items that go in them—are displayed as block elements by default, which means they start on a new line and have some space above and below, but may be altered with CSS. In this section, we'll look at each list type in detail.

Unordered Lists

`<ul>...</ul>`
Unordered list

`<li>...</li>`
List item within an ordered list

Just about any list of examples, names, components, thoughts, or options qualifies as an unordered list. In fact, most lists fall into this category. By default, unordered lists display with a bullet before each list item, but you can change that with a style sheet, as you'll see in a moment.

To identify an unordered list, mark it up as a `ul` element. The opening `<ul>` tag goes before the first list item, and the closing `</ul>` tag goes after the last item. Then, to mark up each item in the list as a list item (`li`), enclose it in opening and closing `<li>` tags, as shown in this example. Notice that there are no bullets in the source document; the browser adds them automatically (FIGURE 5-3):

```
<ul>
  <li>Serif</li>
  <li>Sans-serif</li>
  <li>Script</li>
  <li>Display</li>
  <li>Dingbats</li>
</ul>
```

- Serif
- Sans-serif
- Script
- Display
- Dingbats

FIGURE 5-3. The default rendering of the sample unordered list—the browser adds the bullets automatically

MARKUP TIP

Unordered lists are ideal for marking up links in navigation toolbars.

The only thing that is permitted within an unordered list (that is, between the start and end `<ul>` tags) is one or more list items. You can't put other elements in there, and there may not be any untagged text. However, you can put any type of content element within a list item (`li`).

Nesting Lists

Any list can be nested within another list; it just has to be placed within a list item. Nested lists are commonly used as the markup for hierarchical navigation links. This example shows the structure of an unordered list nested in the second item of an ordered list:

```
<ol>
  <li></li>
  <li>
    <ul>
      <li></li>
      <li></li>
      <li></li>
    </ul>
  </li>
</ol>
```

When you nest an unordered list within another unordered list, the browser automatically changes the bullet style for the second-level list. Unfortunately, the numbering style is not changed by default when you nest ordered lists. You need to set the numbering styles yourself with CSS rules.

But here's the cool part. We can take that same unordered list markup and radically change its appearance by applying different style sheets, as shown in FIGURE 5-4. In the figure, I've turned off the bullets, added my own bullet icons, lined up the items horizontally, and even made them look like graphical buttons. The markup stays exactly the same.

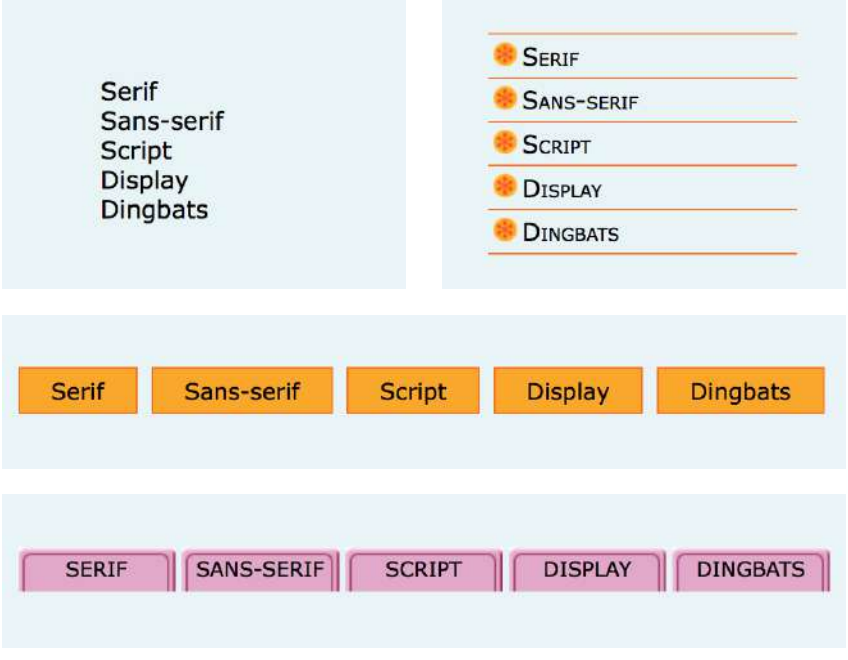


FIGURE 5-4. With style sheets, you can give the same unordered list many looks.

Ordered Lists

`<ol>...</ol>`
Ordered list

`<li>...</li>`
List item within an ordered list

Ordered lists are for items that occur in a particular order, such as step-by-step instructions or driving directions. They work just like the unordered lists described previously, but they are defined with the `ol` element (for "ordered list," of course). Instead of bullets, the browser automatically inserts numbers before ordered list items (see **Note**), so you don't need to number them in the source document. This makes it easy to rearrange list items without renumbering them.

Ordered list elements must contain one or more list item elements, as shown in this example and FIGURE 5-5:

```
<ol>
  <li>Gutenberg develops moveable type (1450s)</li>
  <li>Linotype is introduced (1890s)</li>
  <li>Photocomposition catches on (1950s)</li>
  <li>Type goes digital (1980s)</li>
</ol>
```

NOTE

If something is logically an ordered list, but you don't want numbers to display, remember that you can always remove the numbering with style sheets. So, go ahead and mark up the list semantically as an `ol` *and adjust how it displays with a style rule.*

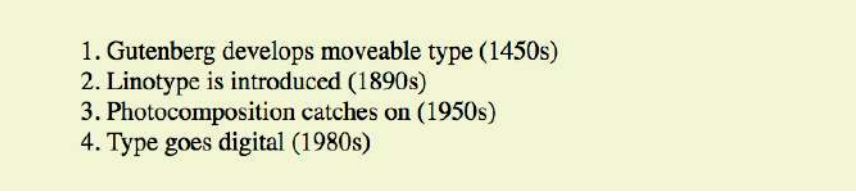

1. Gutenberg develops moveable type (1450s)
2. Linotype is introduced (1890s)
3. Photocomposition catches on (1950s)
4. Type goes digital (1980s)

FIGURE 5-5. The default rendering of an ordered list—the browser adds the numbers automatically

If you want a numbered list to start at a number other than 1, you can use the **start** attribute in the **ol** element to specify another starting number, as shown here:

```
<ol start="17">
  <li>Highlight the text with the text tool.</li>
  <li>Select the Character tab.</li>
  <li>Choose a typeface from the pop-up menu.</li>
</ol>
```

The resulting list items would be numbered 17, 18, and 19, consecutively.

Changing Bullets and Numbering

You can use the **list-style-type** style sheet property to change the bullets and numbers for lists. For example, for unordered lists, you can change the shape from the default dot to a square or an open circle, substitute your own image, or remove the bullet altogether. For ordered lists, you can change the numbers to Roman numerals (I, II, III or i, ii, iii), letters (A, B, C or a, b, c), and several other numbering schemes. In fact, as long as the list is marked up semantically, it doesn't need to display with bullets or numbering at all. Changing the style of lists with CSS is covered in **Chapter 12, Formatting Text**.

Description Lists

Description lists are used for name/value pairs, such as terms and their definitions, questions and answers, or any other types of text and associated information. Their structure is a bit different from the other two lists that we just discussed. The whole description list is marked up as a **dl** element. The content of a **dl** is some number of **dt** elements indicating the names and **dd** elements for their respective values. You might find it helpful to think of them as "terms" (to remember the "t" in **dt**) and "definitions" (for the "d" in **dd**), even though that is only one use of description lists.

<dl>...</dl>
Description list

<dt>...</dt>
A name, such as a term or label

<dd>...</dd>
A value, such as a description or definition

Here is an example of a list that associates forms of typesetting with their descriptions (FIGURE 5-6):

```
<dl>
  <dt>Linotype</dt>
  <dd>Line-casting allowed type to be selected, used, then recirculated
into the machine automatically. This advance increased the speed of
typesetting and printing dramatically.</dd>

  <dt>Photocomposition</dt>
  <dd>Typefaces are stored on film then projected onto photo-sensitive
paper. Lenses adjust the size of the type.</dd>

  <dt>Digital type</dt>
  <dd><p>Digital typefaces store the outline of the font shape in a
format such as Postscript. The outline may be scaled to any size for
output.</p>
      <p>Postscript emerged as a standard due to its support of
graphics and its early support on the Macintosh computer and Apple
laser printer.</p>
  </dd>
</dl>
```

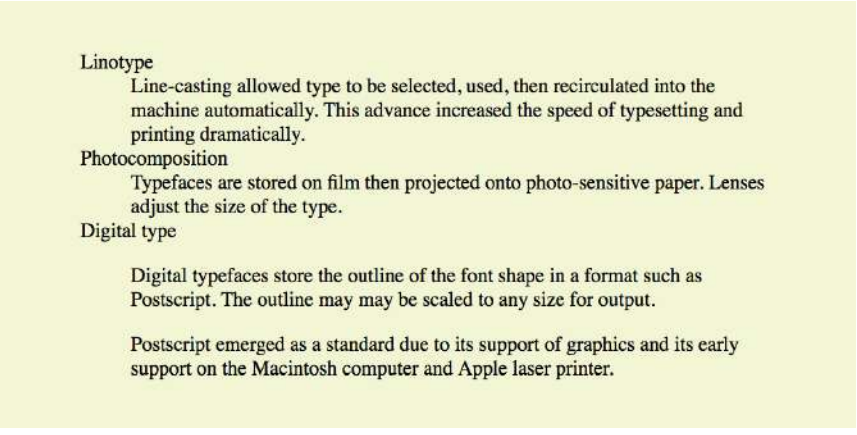

Linotype
Line-casting allowed type to be selected, used, then recirculated into the machine automatically. This advance increased the speed of typesetting and printing dramatically.
Photocomposition
Typefaces are stored on film then projected onto photo-sensitive paper. Lenses adjust the size of the type.
Digital type
Digital typefaces store the outline of the font shape in a format such as Postscript. The outline may may be scaled to any size for output.
Postscript emerged as a standard due to its support of graphics and its early support on the Macintosh computer and Apple laser printer.

FIGURE 5-6. The default rendering of a definition list—definitions are set off from the terms by an indent

The **dl** element is allowed to contain only **dt** and **dd** elements or some number of **div**s (we'll get to the **div** element later in this chapter). You cannot put headings or content grouping elements (like paragraphs) in names (**dt**), but the values (**dd**) can contain any type of flow content. For example, the last **dd** element in the previous example contains two paragraph elements (the awkward default spacing could be cleaned up with a style sheet). It is permitted to have multiple definitions with one term and vice versa.

MORE CONTENT ELEMENTS

`<blockquote>...</blockquote>`
Lengthy, block-level quotation

We've covered paragraphs, headings, and lists, but there are a few more special text elements to add to your HTML toolbox: long quotations (**blockquote**), preformatted text (**pre**), and figures (**figure** and **figcaption**). One thing these elements have in common is that they are categorized as "grouping elements" in the HTML spec, along with **p**, **hr**, the list elements, and **main** and the generic **div**, covered later in this chapter. The other thing they share is that browsers typically display them as block elements by default.

Long Quotations

NOTE

There is also a text-level element, **q**, *for short quotations in the flow of text. I'll talk about that later in this chapter.*

If you have a long quotation, a testimonial, or a section of copy from another source, mark it up as a **blockquote** element (see **Note**). It is recommended that content within **blockquote** elements be contained in other elements, such as paragraphs, headings, or lists, as shown in this example:

```
<p>Renowned type designer, Matthew Carter, has this to say about his
profession:</p>

<blockquote>
  <p>Our alphabet hasn't changed in eons; there isn't much latitude in
what a designer can do with the individual letters.</p>
```

```
    <p>Much like a piece of classical music, the score is written
down. It's not something that is tampered with, and yet, each
conductor interprets that score differently. There is tension in the
interpretation.</p>
</blockquote>
```

FIGURE 5-7 shows the default rendering of the **blockquote** example. This can be altered with CSS.

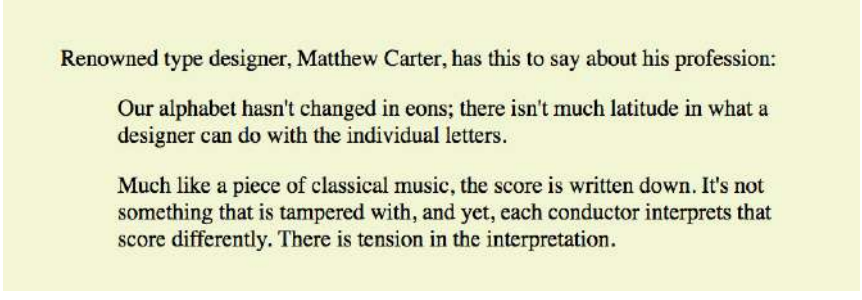

Renowned type designer, Matthew Carter, has this to say about his profession:
Our alphabet hasn't changed in eons; there isn't much latitude in what a designer can do with the individual letters.
Much like a piece of classical music, the score is written down. It's not something that is tampered with, and yet, each conductor interprets that score differently. There is tension in the interpretation.

FIGURE 5-7. The default rendering of a **blockquote** element

Preformatted Text

`<pre>...</pre>`
Preformatted text

In the previous chapter, you learned that browsers ignore whitespace such as line returns and character spaces in the source document. But in some types of information, such as code examples or certain poems, the whitespace is important for conveying meaning. For content in which whitespace is semantically significant, use the preformatted text (**pre**) element. This is a unique element, in that its content is displayed exactly as it is typed—including all the carriage returns and multiple character spaces. By default, preformatted text is also displayed in a constant-width font (one in which all the characters are the same width, also called **monospace**), such as Courier; however, you can easily change the font with a style sheet rule (see **Note**).

NOTE

The **white-space:pre** *CSS property can also be used to preserve spaces and line returns in the source if the effect is stylistic rather than semantically important.*

The **pre** element in this example displays as shown in FIGURE 5-8. The second part of the figure shows the same content marked up as a paragraph (**p**) element for comparison:

```
<pre>
This is                an             example of
      text with a           lot of
                            curious
                            whitespace.
</pre>

<p>
This is                an             example of
      text with a           lot of
                            curious
                            whitespace.
</p>
```

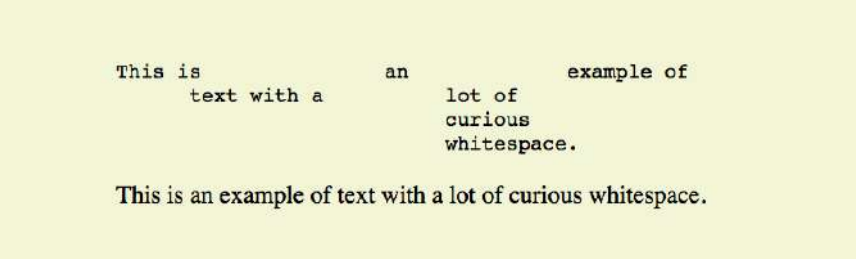


FIGURE 5-8. Preformatted text is unique in that the browser displays the whitespace exactly as it is typed into the source document. Compare it to the paragraph element, in which multiple line returns and character spaces are reduced to a single space.

Figures

`<figure>...</figure>`
Related image or resource

`<figcaption>...</figcaption>`
Text description of a figure

The **figure** element identifies content that illustrates or supports some point in the text. A figure may contain an image, a video, a code snippet, text, or even a table—pretty much anything that can go in the flow of web content. Content in a **figure** element should be treated and referenced as a self-contained unit. That means if a figure is removed from its original placement in the main flow (to a sidebar or appendix, for example), both the figure and the main flow should continue to make sense.

Although you can simply add an image to a page, wrapping it in **<figure>** tags makes its purpose clear semantically. It also works as a hook for applying special styles to figures but not to other images on the page:

```
<figure>
  <img src="piechart.png" alt="chart showing fonts on mobile devices">
</figure>
```

If you want to provide a text caption for the figure, use the **figcaption** element above or below the content inside the **figure** element, as shown here—this is a more semantically rich way to mark up the caption than using a simple **p** element:

```
<figure>
  <pre>
    <code>
      body {
        background-color: #000;
        color: yellow;
      }
    </code>
  </pre>
  <figcaption>Sample CSS rule.</figcaption>
</figure>
```

In EXERCISE 5-1, you'll get a chance to mark up a document yourself and try out the basic text elements we've covered so far.

EXERCISE 5-1. Marking up a recipe

The owners of the Black Goose Bistro have decided to share recipes and news on their site. In the exercises in this chapter, we'll assist them with content markup.

In this exercise, you will start with the raw text of a recipe. It's up to you to decide which element is the best semantic match for each chunk of content. You'll use paragraphs, headings, lists, and at least one special content element.

You can write the tags right on this page. Or, if you want to use a text editor and see the results in a browser, this text file, as well as the final version with markup, is available at *learningwebdesign.com/6e/materials*.

```
Tapenade (Olive Spread)

This is a really simple dish to prepare and it's always a big hit at
parties. My father recommends:

"Make this the night before so that the flavors have time to blend. Just
bring it up to room temperature before you serve it. In the winter, try
serving it warm."

Ingredients

1 8oz. jar sundried tomatoes
2 large garlic cloves
2/3 c. kalamata olives
1 t. capers

Instructions
Combine tomatoes and garlic in a food processor. Blend until as smooth
as possible.

Add capers and olives. Pulse the motor a few times until they are
incorporated, but still retain some texture.

Serve on thin toast rounds with goat cheese and fresh basil garnish
(optional).
```

ORGANIZING PAGE CONTENT

So far, the elements we've covered handle very specific tidbits of content: a paragraph, a heading, a figure, and so on. Prior to HTML5, there was no way to group these bits into larger parts other than wrapping them in a generic division (**`div`**) element (we'll cover **`div`** in more detail later). HTML5 introduced new elements that give semantic meaning to sections of a typical web page or application (see **Note**), including main content (**`main`**), headers (**`header`**), footers (**`footer`**), sections (**`section`**), articles (**`article`**), navigation (**`nav`**), and tangentially related or complementary content (**`aside`**). These elements (with the exception of **`main`**, which is considered grouping content) are categorized as **sectioning elements** in the HTML spec. Curiously, the spec lists the old **`address`** element as a sectioning element as well, so we'll look at that one here too.

NOTE

The element names are based on a Google study in 2005 that looked at the top 20 names that developers assigned to generic division (`div`) elements.

Main Content

`<main>...</main>`
Primary content area of page or app

Web pages these days are loaded with different types of content: mastheads, sidebars, ads, footers, more ads, even more ads, and so on. Explicitly pointing out the primary content of a page or application with the **main** element helps screen readers and other assistive technologies know where the main content of the page begins and replaces the "Skip to main content" links that have been utilized in the past.

The content of a **main** element should be unique to that page. In other words, headers, sidebars, and other elements that appear across multiple pages in a site should not be included in the **main** section:

```
<body>
  <header>...</header>
  <main>
    <h1>Humanist Sans Serif</h1>
    <!-- code continues -->
  </main>
</body>
```

The HTML specification states that pages should have only one **main** section and that it should not be nested within an **article**, **aside**, **header**, **footer**, or **nav**. Doing so will cause the document to be invalid.

Headers and Footers

`<header>...</header>`
Introductory material for page, section, or article

`<footer>...</footer>`
Footer for page, section, or article

Because web authors had been labeling header and footer sections in their documents for years, it was obvious that the full-fledged **header** and **footer** elements added in HTML5 would come in handy. Let's start with headers.

Headers

The **header** element is used for introductory material that typically appears at the beginning of a web page or at the top of a section or article (we'll get to those elements soon). There is no specified list of what a **header** must or should contain; anything that makes sense as the introduction to a page or section is acceptable. In the following example, the document header includes a logo image, the site title, and navigation:

```
<body>
  <header>
    <img src="/images/logo.png" alt="logo">
    <h1>Nuts about Web Fonts</h1>
    <nav>
      <ul>
        <li><a href="/index.html">Home</a></li>
        <li><a href="/blog.html">Blog</a></li>
        <li><a href="/shop.html">Shop</a></li>
      </ul>
    </nav>
  </header>
  <!--page content-->
</body>
```

NOTE

The `<a href=""></a>` *code in the examples is the markup for adding links to other web pages. We'll take on links in* **Chapter 6, Adding Links**. *The value is the URL for the page, as shown here, but I'll be using a simple slash as a space-saving measure going forward.*

When used in an individual article, the **header** might include the article title, author, and publication date, as shown here:

```
<article>
  <header>
    <h1>More about WOFF</h1>
    <p>by Jennifer Robbins, <time datetime="2023-11-11">November 11,
2023</time></p>
  </header>
  <!-- article content here -->
</article>
```

NOTE

The **time** *element will be discussed in the section* **"Dates and times"** *later in this chapter.*

Footers

The **footer** element is used to indicate the type of information that typically comes at the end of a page or an article, such as its author, copyright information, or navigation. The **footer** element may apply to the entire document, or it could be associated with a particular section or article. If the footer is contained directly within the **body** element, either before or after all the other **body** content, then it applies to the entire page or application. If it is contained in a sectioning element (**section**, **article**, **nav**, or **aside**), it is parsed as the footer for just that section. Note that although it is called "footer," there is no requirement that it appear last in the document or sectioning element. It could also appear at or near the beginning, if that makes sense.

NOTE

Neither **header** *nor* **footer** *elements are permitted to contain nested* **header** *or* **footer** *elements.*

In this simple example, we see the typical information listed at the bottom of an article marked up as a **footer**:

```
<article>
  <header>
    <h1>More about WOFF</h1>
    <p>by Jennifer Robbins, <time datetime="2024-10-07">October 7,
2024</time></p>
  </header>
  <!-- article content here -->
  <footer>
    <p><small>Copyright ©2025 Jennifer Robbins</small></p>
    <nav>
      <ul>
        <li><a href="/">Previous</a></li>
        <li><a href="/">Next</a></li>
      </ul>
    </nav>
  </footer>
</article>
```

Sections and Articles

Long documents are easier to use when they are divided into smaller parts. For example, books are divided into chapters, and newspapers have sections for local news, sports, comics, and so on. To divide long web documents into thematic sections, use the aptly named **section** element. Sections typically include a heading (inside the **section** element) plus content that has a meaningful reason to be grouped together.

<section>...</section>
Thematic group of content

<article>...</article>
Self-contained, reusable composition

NOTE

The HTML spec recommends using the generic **div** *element instead of* **section** *if the purpose for grouping the elements is simply to provide a hook for styling.*

The **section** element has a broad range of uses, from dividing a whole page into major sections to identifying thematic sections within a single article. A **section** should include a heading (**h1**–**h6**) that identifies the content, although it is not strictly required in order to be valid. In the following example, a document with information about typography resources has been divided into two sections based on resource type:

```
<section>
  <h2>Typography Books</h2>
  <ul>
     <li>...</li>
  </ul>
</section>

<section>
  <h2>Online Tutorials</h2>
  <p>These are the best tutorials on the web.</p>
  <ul>
    <li>...</li>
  </ul>
</section>
```

Use the **article** element for self-contained works that could stand alone or be reused in a different context (such as syndication). It is useful for magazine or newspaper articles, blog posts, comments, or other items that could be extracted for external use. You can think of it as a specialized **section** element that answers "yes" to the question "Could this appear on another site and make sense?"

A long **article** could be broken into a number of sections, as shown here:

```
<article>
  <h1>Get to Know Helvetica</h1>
  <section>
    <h2>History of Helvetica</h2>
    <p>...</p>
  </section>

  <section>
    <h2>Helvetica Today</h2>
    <p>...</p>
  </section>
</article>
```

Conversely, a **section** in a web document might be composed of a number of articles:

```
<section id="essays">
  <h1>For Further Reading</h1>
  <article>
    <h1>A Fresh Look at Futura</h1>
    <p>...</p>
  </article>
  <article>
    <h1>Getting Personal with Humanist</h1>
    <p>...</p>
  </article>
</section>
```

The **section** and **article** elements are easily confused, particularly because it is possible to nest one in the other and vice versa. Keep in mind that if the content is self-contained and could appear outside the current context, it is best marked up as an **article**.

Aside (Sidebars)

`<aside>...</aside>`
Tangentially related material

The **aside** element identifies content that is separate from, but tangentially related to, the surrounding content. In print, its equivalent is a sidebar, but the spec doesn't call it "sidebar" because putting something on the "side" is a presentational description, not semantic. Nonetheless, a sidebar is a good mental model for using the **aside** element. **aside** can be used for pull quotes, background information, lists of links, callouts, or anything else that might be associated with (but not critical to) a document.

Here, an **aside** element is used for a list of links related to the main article:

```
<h1>Web Typography</h1>
<p>Back in 1997, there were competing font formats and tools for
making them...</p>
<p>We now have a number of methods for using beautiful fonts on web
pages...</p>
<aside>
  <h2>Web Font Resources</h2>
  <ul>
    <li><a href="http://variablefonts.io">Variable Fonts</a></li>
    <li><a href="http://fonts.google.com">Google Fonts</a></li>
  </ul>
</aside>
```

The **aside** element has no default rendering, so you will need to set its display type to adjust its appearance and layout with style sheet rules.

Navigation

`<nav>...</nav>`
Primary navigation links

The **nav** element gives developers a semantic way to identify navigation for a site. Earlier in this chapter, we saw an unordered list that might be used as the top-level navigation for a font catalog site. Wrapping that list in a **nav** element describes its purpose accurately:

```
<nav>
  <ul>
    <li><a href="/serif.html">Serif</a></li>
    <li><a href="/sansserif.html">Sans-serif</a></li>
    <li><a href="/script.html">Script</a></li>
    <li><a href="/display.html">Display</a></li>
    <li><a href="/dingbats.html">Dingbats</a></li>
  </ul>
</nav>
```

Not all lists of links should be wrapped in a **nav** element, however. The spec makes it clear that **nav** should be used for links that provide primary navigation around a site or a lengthy section or article. The **nav** element may be especially helpful from an accessibility perspective.

Search-Related Content

`<search>...</search>`
Content related to searching or filtering

Just when you thought we had all the text elements we'd ever need, in 2023 the **search** grouping element was born! The **search** element is used to indicate a part of a page that contains form controls related to searching or filtering content. The searching functionality itself is handled by a **form** and its **input** elements, but marking up that form with the **search** element can make the search bar easier to find with assistive devices.

search is similar to **main** or **nav** in that its job is to add semantic meaning that can be helpful for accessibility. In fact, it was added to the spec in order to map to the "search" landmark role in ARIA (ARIA and roles are explained in the **"Improving Accessibility with ARIA"** section later in this chapter).

The **search** element has only been supported in browsers since 2023, but non-supporting browsers will just ignore it, so there is no harm adding it today.

Addresses

`<address>...</address>`
Contact information

Last and, well, least is the **address** element that is used to create an area for contact information for the author or maintainer of the document. It is generally placed at the end of the document or in a section or article within a document. An **address** would be right at home in a **footer** element. It is important to note that the **address** element should *not* be used for any other type of address on a page, such as a business's mailing address. It is intended specifically for author contact information (although that could potentially be a mailing address). Following is an example of its intended use:

```
<address>
  <p>Contributed by <a href="../authors/robbins/">Jennifer Robbins</a>,
  <a href="http://www.oreilly.com/">O'Reilly Media</a> </p>
</address>
```

THE TEXT-LEVEL ELEMENT ROUNDUP

Now that we've identified the larger chunks of content, we can provide semantic meaning to phrases within the chunks by using what the HTML specification calls **text-level semantic elements**. You may also hear them called **inline elements**, as that was the term used in earlier HTML specifications (because they display in the flow of text by default and do not cause any line breaks).

Text-Level Elements

Despite all the types of information you could add to a document, there are only a couple of dozen text-level semantic elements. TABLE 5-1 lists all of them.

TABLE 5-1. Text-level semantic elements

Element	Description
a	An anchor or hypertext link (see **Chapter 6** for details)
abbr	Abbreviation
b	Added visual attention, such as keywords (bold)
bdi	Indicates text that may have directional requirements
bdo	Bidirectional override; explicitly indicates text direction (left to right, **ltr**, or right to left, **rtl**)
br	Line break
cite	Citation; a reference to the title of a work, such as a book title
code	Computer code sample
data	Machine-readable equivalent dates, time, weights, and other measurable values
del	Deleted text; indicates an edit made to a document
dfn	The defining instance or first occurrence of a term
em	Emphasized text
i	Alternative voice (italic) or alternate language
ins	Inserted text; indicates an insertion in a document
kbd	Keyboard; text entered by a user (for technical documents)
mark	Contextually relevant text
menu	Represents a toolbar consisting of its contents, in the form of an unordered list
q	Short, text-level quotation
ruby, rt, rp	Provides annotations or pronunciation guides under East Asian typography and ideographs
s	Incorrect text (strike-through)
samp	Sample output from programs
small	Small print, such as a copyright or legal notice (usually displayed in a smaller type size by default)
span	Generic phrase content
strong	Content of strong importance
sub	Subscript
sup	Superscript
time	Machine-readable time data
u	Indicates a formal name, misspelled word, or text that would be underlined
var	A variable or program argument (for technical documents)
wbr	Word break

The Text-Level Elements Backstory

Many of the text-level elements that have been around since the dawn of the web were introduced to change the visual formatting of text selections because of the lack of a style sheet system. If you wanted bolded text, you marked it as **b**. Italics? Use the **i** element. In fact, there was once a **font** element used solely to change the font, color, and size of text (the horror!).

Not surprisingly, HTML5 kicked the purely presentational **font** element to the curb. However, many of the old-school presentational text-level elements (for example, **u** for underline and **s** for strikethrough) were kept and given new semantic definitions (**b** is now for "added visual attention," **s** for "inaccurate text").

Many text-level elements have the expected style rendering (bold for the **b** element, for example). Others are purely semantic (such as **abbr** or **time**) and don't have default renderings. For any text-level elements, you can use CSS rules if you want to change the way they display.

HTML5 also declared the following text-level elements obsolete: **acronym**, **applet**, **basefont**, **big**, **center**, **dir** (directory), **font**, **isindex** (search box), **strike**, and **tt** (teletype). I mention them here in case you run across them when viewing the source of an old document or if you are using an older web authoring tool. You should not use them today.

Although it may be useful having all of the text-level elements listed together in a table, they certainly deserve more detailed explanations. Let's take a closer look at each of these elements and what they're used for.

Emphasized text

`<em>...</em>`
Stressed emphasis

Use the **em** element to indicate which part of a sentence should be stressed or emphasized. The placement of **em** elements affects how a sentence's meaning is interpreted. Consider the following sentences, which are identical except for which words are stressed:

```
<p><em>Arlo</em> is very smart.</p>
<p>Arlo is <em>very</em> smart.</p>
```

The first sentence indicates *who* is very smart. The second example is about *how* smart he is. Notice that the **em** element has an effect on the meaning of the sentence.

Emphasized text (**em**) elements nearly always display in italics by default (FIGURE 5-9), but of course you can make them display any way you like with a style sheet. Screen readers may use a different tone of voice to convey stressed content, which is why you should use an **em** element only when it makes sense semantically, not just to achieve italic text.

Important text

`<strong>...</strong>`
Strong importance

The **strong** element indicates that a word or phrase is important, serious, or urgent. In the following example, the **strong** element identifies the portion of the instructions that requires extra attention. The **strong** element does not change the meaning of the sentence; it merely draws attention to the important parts:

```
<p>When returning the car, <strong>drop the keys in the red box by the
front desk</strong>.</p>
```

Visual browsers typically display **strong** text elements in bold text by default (FIGURE 5-9). Screen readers may use a distinct tone of voice for important content, so mark text as **strong** only when it makes sense semantically, not just to make it bold.

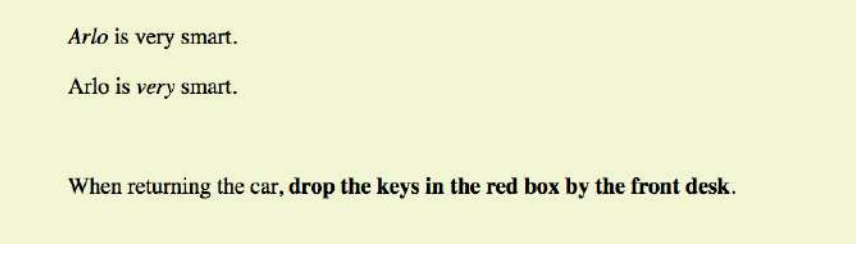


FIGURE 5-9. The default rendering of emphasized and strong text

Elements originally named for their presentational properties

As long as we're talking about bold and italic text, let's see what the old **b** and **i** elements are up to now. The elements **b**, **i**, **u**, **s**, and **small** were introduced in the old days of the web as a way to provide typesetting instructions (bold, italic, underline, strike-through, and smaller text, respectively). Despite their original presentational purposes, these elements are still around in the current HTML spec, with updated, semantic definitions. Browsers still render them by default as you'd expect (FIGURE 5-10 shows how the examples given here are typically displayed). However, if a type style change is all you're after, using a style sheet rule is the appropriate solution. Save these for when they are semantically appropriate.

Let's look at these elements and their correct usage, as well as the style sheet alternatives.

`<b>...</b>`
Keywords or visually emphasized text (bold)

`<i>...</i>`
Alternative voice (italic)

`<s>...</s>`
Incorrect text (strike-through)

`<u>...</u>`
Annotated text (underline)

`<small>...</small>`
Legal text; small print (smaller type size)

b

Indicates keywords, product names, or other phrases that need to stand out from the surrounding text without conveying added importance or emphasis (see **Note**). [*Old definition:* Bold.]

CSS property: For bold text, use **`font-weight: bold`**

Example:
```
<p>The slabs at the ends of letter strokes are called
<b>serifs</b>.</p>
```

i

Indicates text that is in a different voice or mood than the surrounding text, such as a phrase from another language, a technical term, or a thought. [*Old definition:* Italic.]

CSS property: For italic text, use **`font-style: italic`**

Example:
```
<p>Simply change the font and <i>Voila!</i>, a new
personality!</p>
```

s

Indicates text that is incorrect. [*Old definition:* Strike-through text.]

CSS property: To draw a line through a selection of text, use **`text-decoration: line-through`**

Example:
```
<p>Scala Sans was designed by <s>Eric Gill</s> Martin
Majoor.</p>
```

u

Indicates text that should have some sort of nontext annotation supplied (such as an underline). There are a few instances when underlining has

NOTE

It helps me to think about how a screen reader would read the text. If I don't want the word read in a loud, emphatic tone of voice, but it really should be bold, then **b** *may be more appropriate than* **strong**.

semantic significance, such as underlining a formal name in Chinese or indicating a misspelled word after a spell check, such as the misspelled "Helvitica" in the following example. Note that underlined text is easily confused with a link and should generally be avoided, except for a few niche cases. [*Old definition:* Underline.]

CSS property: For underlined text, use **`text-decoration: underline`**

Example:
```
<p>New York subway signage is set in <u>Helviteca</u>.</p>
```

`small`

Indicates an addendum or side note to the main text, such as the legal "small print" at the bottom of a document. [*Old definition:* Renders in font smaller than the surrounding text.]

CSS property: To make text smaller, use **`font-size`** (example: **`font-size: 80%`**)

Example:
```
<p><small>(This font is free for personal and commercial use.)</small></p>
```

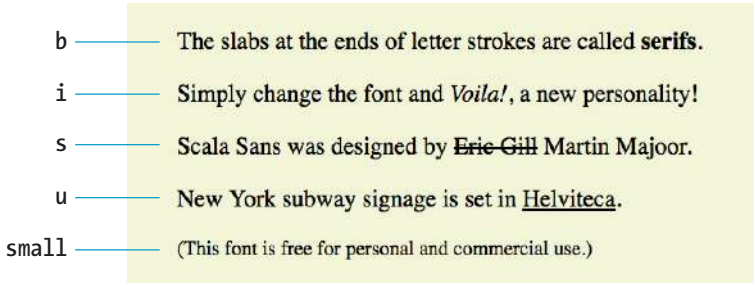

b
The slabs at the ends of letter strokes are called serifs.
i
Simply change the font and Voila!, a new personality!
s
Scala Sans was designed by Eric Gill Martin Majoor.
u
New York subway signage is set in Helviteca.
small
(This font is free for personal and commercial use.)

FIGURE 5-10. The default rendering of `b`, `i`, `s`, `u`, and `small` elements

Short quotations

`<q>...</q>`
Short text-level quotation

Use the quotation (**`q`**) element to mark up short quotations, such as "To be or not to be," in the flow of text, as shown in this example (FIGURE 5-11):

```
Matthew Carter says, <q>Our alphabet hasn't changed in eons.</q>
```

According to the HTML spec, browsers should add quotation marks around **`q`** elements automatically, so you don't need to include them in the source document. Most major browsers—like Firefox, Chrome, and Microsoft Edge—render curly quotes, which is preferable. Safari (which I used for this example) renders them as straight quotes, as shown in FIGURE 5-11.

The **`q`** element has an optional **`cite`** attribute for listing a URL for the source of the quote. The spec makes it clear that marking up short quotations with **`q`** is optional; they can also be conveyed by typing quotation marks right in the text.

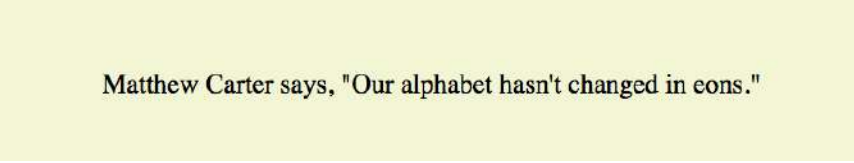


FIGURE 5-11. Browsers add quotation marks automatically around q elements.

Abbreviations and acronyms

`<abbr>...</abbr>`
Abbreviation or acronym

Marking up acronyms and abbreviations with the **abbr** element provides useful information for search engines, screen readers, and other devices. Abbreviations are shortened versions of a word ending in a period ("Conn." for "Connecticut," for example). Acronyms are abbreviations generally formed by the first letters of the words in a phrase (such as NASA or USA). The **title** attribute provides the long version of the shortened term, as shown in this example:

```
<abbr title="Points">pts.</abbr>
<abbr title="American Type Founders">ATF</abbr>
```

Citations

`<cite>...</cite>`
Citation

The **cite** element is used to identify a reference to another document, such as a book, magazine, article title, and so on. Citations are typically rendered in italic text by default. Here's an example:

```
<p>Passages of this article were inspired by <cite>The Complete Manual
of Typography</cite> by James Felici.</p>
```

Defining terms

`<dfn>...</dfn>`
Defining term

It is common to point out the first and defining instance of a word in a document in some fashion. In this book, defining terms are set in blue text. In HTML, you can identify them with the **dfn** element, as shown here, and format them visually using style sheets:

```
<p><dfn>Script typefaces</dfn> are based on handwriting.</p>
```

Nesting Elements

You can apply two elements to a string of text (for example, a phrase that is both a quote and in another language), but be sure they are nested properly. That means the inner element, including its closing tag, must be completely contained within the outer element:

```
<q><i>Je ne sais pas.</i></q>
```

Here is an example of elements that are nested incorrectly. Notice that the inner **i** element is not closed within the containing **q** element:

```
<q><i>Je ne sais pas.</q></i>
```

It's easy to spot the nesting error in an example that is this short, but when you're nesting long passages or nesting multiple levels deep, it's easy to end up with overlaps. One advantage to using an HTML code editor is that it can automatically close elements for you correctly, or point out when you've made a mistake.

Program code elements

`<code>...</code>`
Code

`<var>...</var>`
Variable

`<samp>...</samp>`
Program sample

`<kbd>...</kbd>`
User-entered keyboard strokes

A number of text-level elements are used for describing the parts of technical documents, such as code (**code**), variables (**var**), program samples (**samp**), and user-entered keyboard strokes (**kbd**). It's a quaint reminder of HTML's origins in the scientific world (Tim Berners-Lee developed HTML to share documents at the CERN particle physics lab in 1989).

Code, sample, and keyboard elements typically render in a constant-width (also called **monospace**) font such as Courier by default. Variables usually render in italics.

Subscript and superscript

`<sub>...</sub>`
Subscript

`<sup>...</sup>`
Superscript

The subscript (**sub**) and superscript (**sup**) elements cause the selected text to display in a smaller size, positioned slightly below (**sub**) or above (**sup**) the baseline. These elements may be helpful for indicating chemical formulas or mathematical equations.

FIGURE 5-12 shows how these examples of subscript and superscript typically render in a browser:

```
<p>H<sub>2</sub>0</p>

<p>E=MC<sup>2</sup></p>
```

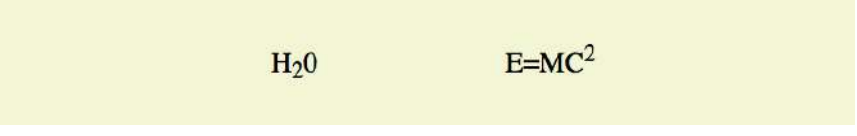


FIGURE 5-12. Subscript and superscript

Highlighted text

`<mark>...</mark>`
Contextually relevant text

The **mark** element indicates a word that may be considered especially relevant to the reader. One might use it to dynamically highlight a search term in a page of results, to manually call attention to a passage of text, or to indicate the current page in a series. Some designers (and browsers) give marked text a light-colored background, as though it were marked with a highlighter marker, as shown in FIGURE 5-13:

```
<p> ... PART I. ADMINISTRATION OF THE GOVERNMENT. TITLE IX.
TAXATION. CHAPTER 65C. MASS. <mark>ESTATE TAX</mark>. Chapter 65C:
Sect. 2. Computation of <mark>estate tax</mark>.</p>
```

... PART I. ADMINISTRATION OF THE GOVERNMENT. TITLE IX. TAXATION. CHAPTER 65C. MASS. ESTATE TAX. Chapter 65C: Sect. 2. Computation of estate tax.

FIGURE 5-13. In this example, search terms are identified with `mark` elements and given a yellow background with a style sheet so they are easier for the reader to find.

Dates and times

`<time>...</time>`
Time data

When we look at the phrase "noon on November 4," we know that it is a date and a time. But the context might not be so obvious to a computer program. The **`time`** element allows us to mark up dates and times in a way that is comfortable for a human to read, but also encoded in a standardized way that computers can use (see **Note**). The content of the element presents the information to people, and the **`datetime`** attribute presents the same information in a machine-readable way.

NOTE

The `time` *element is not intended for marking up times for which a precise time or date cannot be established, such as "the end of last year" or "the turn of the century."*

The **`time`** element indicates dates, times, or date/time combos. It might be used, for example, to pass the date and time information to an application (such as for saving an event to a personal calendar), or by search engines to find the most recently published articles, or to restyle time information into an alternate format (e.g., changing 18:00 to 6 p.m.).

The **`datetime`** attribute specifies the date and/or time information in a standardized time format illustrated in FIGURE 5-14. The full time format begins with the date (year–month–day). The time section begins with a letter "T" and lists hours (on the 24-hour clock), minutes, seconds (optional), and milliseconds (also optional). Finally, the time zone is indicated by the number of hours behind (–) or ahead of (+) Greenwich Mean Time (GMT). For example, "–05:00" indicates the Eastern Standard time zone, which is five hours behind GMT. For dates and times alone, you can omit the other sections.

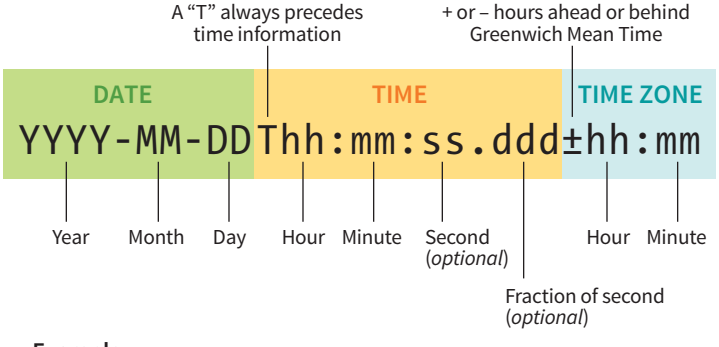


FIGURE 5-14. Standardized date and time syntax

Here are a few examples of valid values for **datetime**:

- **Time only:** 9:30 p.m.

```
<time datetime="21:30">9:30 p.m.</time>
```

- **Date only:** June 19, 2016

```
<time datetime="2016-06-19">June 19, 2016</time>
```

- **Date and time:** Sept. 5, 1970, 1:11 a.m.

```
<time datetime="1970-09-05T01:11:00">Sept. 5, 1970, 1:11 a.m.</time>
```

- **Date and time, with time zone information:** 8:00 a.m. on July 19, 2023, in Providence, RI

```
<time datetime="2023-07-19T08:00:00-05:00">July 19, 2023, 8 a.m.,
Providence, RI</time>
```

NOTE

You can also use the **time** *element without the* **datetime** *attribute, but its content must be a valid date/time string.*

```
<time>2025-04-23</time>
```

■ FURTHER READING

For more information on the intricate ins and outs of specifying dates and times, with examples, check out the **time** element entry in the HTML specification (*html.spec.whatwg.org/multipage/text-level-semantics.html#the-time-element*).

Machine-readable information

`<data>...</data>`
Machine-readable data

The **data** element is another tool for helping computers make sense of content. It can be used for all sorts of data, including dates, times, measurements, weights, microdata, and so on. The required **value** attribute provides the machine-readable information. Here are a couple of examples:

```
<data value="12">Twelve</data>

<data value="978-1-449-39319-9">CSS: The Definitive Guide</data>
```

I'm not going to go into more detail on the **data** element, because as a beginner, you are unlikely to be dealing with machine-readable data quite yet. But it is interesting to see how markup can be used to provide usable information to computer programs and scripts as well as to your fellow humans.

Inserted and deleted text

`<ins>...</ins>`
Inserted text

`<del>...</del>`
Deleted text

The **ins** and **del** elements are used to mark up edits indicating parts of a document that have been inserted or deleted (respectively):

```
Chief Executive Officer: <del title="retired">Peter Pan</del><ins>Pippi
Longstocking</ins>
```

These elements rely on style rules for presentation (i.e., there is no dependable browser default). Both the **ins** and **del** elements can contain either text-level or grouping elements, depending on what type of content they contain.

Adding Breaks

HTML provides elements for semantically inserting breaks in lines and words.

Line breaks

`<br>`
Line break

Occasionally, you may need to add a line break within the flow of text. We've seen how browsers ignore line breaks in the source document, so we need a specific directive to tell the browser to "add a line break here."

The line break element (**br**) does exactly that. It can be used, for example, to break up lines of addresses or poetry. **br** is a void element, which means it does not have content. Add the element in the flow of text where you want a break to occur, as shown here and in FIGURE 5-15:

```
<p>So much depends <br>upon <br><br>a red wheel <br>barrow</p>
```

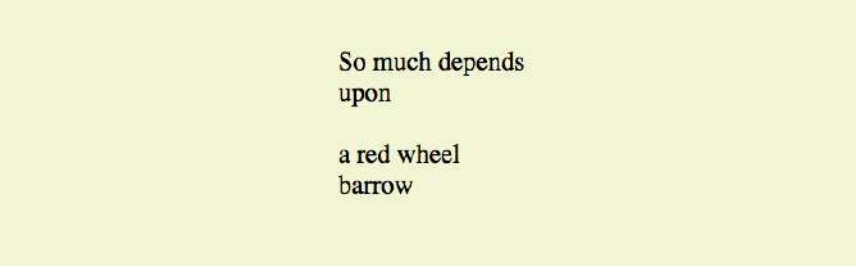


FIGURE 5-15. Line breaks are inserted at each **br** element (example extracted from "The Red Wheelbarrow" by William Carlos Williams).

Unfortunately, the **br** element is easily abused. Be careful that you aren't using **br** elements to force breaks into text that really ought to be a list. For example, don't do this:

```
<p>Times<br>
Georgia<br>
Garamond
</p>
```

If it's a list, use the semantically correct unordered list element instead, and turn off the bullets with style sheets:

```
<ul>
  <li>Times</li>
  <li>Georgia</li>
  <li>Garamond</li>
</ul>
```

Word breaks

`<wbr>`
Word break

The word break (**wbr**) element lets you mark the place where a word should break (a "line break opportunity," according to the spec) should there not be enough room for the whole word (FIGURE 5-16). It takes some of the guesswork away from the browser and allows authors to control the best spot for the word to be split over two lines (without **wbr**, the word stays together, and if there is not enough room, the whole word wraps to the next line). If there is enough room, the word stays in one piece. Note that the browser does not add a hyphen when the word breaks over two lines (see **Note**). The **wbr** behaves as though it were a character space in the middle of the word:

```
<p>The biggest word you've ever heard and this is how it goes:
<em>supercali<wbr>fragilistic<wbr>expialidocious</em>!</p>
```

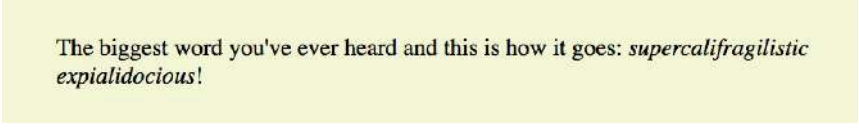


FIGURE 5-16. When there is not enough room for a word to fit on a line, it will break at the location of a **wbr** element.

NOTE

If you'd like a hyphen to appear when the word breaks onto two lines, you can insert a soft hyphen character (``) at the spot you'd like the word to break. If the word stays on one line, the soft hyphen stays hidden, but if the word breaks, a hyphen character displays at the end of the line. Character entities like `` are discussed at the end of this chapter.

You've been introduced to 32 new elements since your last exercise, so I say it's time to give some of the text-level elements a try in EXERCISE 5-2.

Accommodating Non-Western Languages

If the web is to reach a truly worldwide audience, it needs to be able to support the display of all the languages of the world, with all their unique alphabets, symbols, directionality, and specialized punctuation. The W3C's efforts for internationalization (often referred to as "i18n") ensure that web technologies are usable worldwide.

Internationalization efforts include the following:

- Using the Unicode character encoding, which contains the characters, glyphs, symbols, ideographs, and the like from all active, modern languages.
- Declaring the primary language of a document by using a two-letter language code from the ISO 639-1 standard (available at *www.loc.gov/standards/iso639-2/php/code_list.php*). For example, English is "EN," Czech is "CS," and German is "DE." Use the **lang** attribute in the **html** element to declare the language for the whole document, or in individual elements that require clarification.
- Accommodating the various writing directions of languages. In HTML, the **dir** attribute explicitly sets the direction for the document or an element to **ltr** (left to right) or **rtl** (right to left). On phrase-level elements, it prevents text within the element from influencing the ordering of text outside it.

  For example, to include a passage of Hebrew in an English document, use the **dir** attribute to indicate that the phrase should be displayed right to left:

  ```
  <p>This is how you write Shalom: <span
  dir="rtl">שלום</span></p>
  ```

- Providing a system that allows for **ruby annotation**, notes that typically appear above ideographs from East Asian languages to give pronunciation clues or translations (**ruby**, **rt**, and **rp** elements). See the spec for details if this is something you need to do.
- Optionally providing alternative links to translated versions of your page, using the **link** element with the **hreflang** attribute in the **head** of the document. For example, this **link** element points to a separate version of your page, translated into German:

  ```
  <link rel="alternate" title="Das Seite"
  href="https://example.com/de/page" hreflang="de">
  ```

The W3C internationalization techniques page (*www.w3.org/International/techniques/authoring-html*) provides a thorough collection of HTML and CSS authoring techniques and resources to help with your internationalization efforts.

EXERCISE 5-2. Identifying text-level elements

This little post for the Black Goose Bistro News page will give you an opportunity to identify and mark up a variety of text-level elements. See if you can find phrases to mark up accurately with the following elements:

```
b   br   cite   dfn   em   i   q   small   time
```

Because markup is always somewhat subjective, your resulting markup may not look exactly like my final markup, but there is an opportunity to use all of the preceding elements in the article. For extra credit, there is a phrase that could have two elements applied to it. (Hint: look for a term in another language.) Remember to nest them properly by closing the inner element before you close the outer one.

You can write the tags right on this page. Or, if you want to use a text editor and see the results in a browser, this text file is available online at *learningwebdesign.com/6e/materials* along with the resulting code:

```
<article>

<header>
<p>posted by BGB, November 15, 2024</p>
</header>

<h2>Low and Slow</h2>
<p>This week I am extremely excited about a new cooking technique called
sous vide. In sous vide cooking, you submerge the food (usually vacuum-
sealed in plastic) into a water bath that is precisely set to the target
temperature you want the food to be cooked to. In his book, Cooking for
Geeks, Jeff Potter describes it as ultra-low-temperature poaching.</p>
<p>Next month, we will be serving Sous Vide Salmon with Dill
Hollandaise. To reserve a seat at the chef table, contact us before
November 30.</p>

<p>blackgoose@example.com
555-336-1800</p>

<p>Warning: Sous vide cooked salmon is not pasteurized. Avoid it if you
are pregnant or have immunity issues.</p>

</article>
```

GENERIC ELEMENTS (DIV AND SPAN)

What if none of the elements we've talked about so far accurately describes your content? After all, there are endless types of information in the world but, as you've seen, not all that many semantic elements. Fortunately, HTML provides two generic elements that can be customized to describe your content perfectly. The **div** element indicates a division of content, and **span** indicates a word or phrase for which no text-level element currently exists. The generic elements are given meaning and context with the **id** and **class** attributes, which we'll discuss in a moment.

`<div>...</div>`
Generic block-level element

`<span>...</span>`
Generic text-level element

The **div** and **span** elements have no inherent presentation qualities of their own, but you can use style sheets to format them however you like. In fact, generic elements are a primary tool in standards-based web design because they enable authors to accurately describe content and offer plenty of "hooks" for adding style rules. They also allow elements on the page to be accessed and manipulated by JavaScript.

We're going to spend a little time on **div** and **span** elements, as well as the **id** and **class** attributes, to learn how authors use them to structure content.

Divide It Up with a div

The **div** element is used to create a logical grouping of content or elements on the page. It indicates that its content belongs together in a conceptual unit or should be treated as a unit by CSS or JavaScript. By marking related content as a **div** and giving it a unique **id** or indicating that it is part of a **class**, you give context to the elements in the grouping. The use of **div**s to structure content for styling, layout, and scripting is a key feature of modern web development, so chances are you'll be using a lot of them.

Let's look at a few examples of **div** elements.

In this example, a **div** element is used as a container to group an image and two paragraphs into a product "listing":

```
<div class="listing">
  <img src="images/felici-cover.gif" alt="">
  <p><cite>The Complete Manual of Typography</cite>, James Felici</p>
  <p>A combination of type history and examples of good and bad type
design.</p>
</div>
```

By putting those elements in a **div**, we've made it clear that they are conceptually related. It also allows us to style **p** elements within listings differently than other **p** elements in the document.

Another common use of a **div** is to break a page into sections for layout purposes. In this example, a heading and several paragraphs are enclosed in a **div** and identified as the "news" division:

```
<div id="news">
  <h1>New This Week</h1>
  <p>We've been working on...</p>
  <p>And last but not least, ...</p>
</div>
```

Now we have a custom element that we've given the name "news." You might be thinking, "Hey, couldn't you use a **section** element for that?" You could! In fact, authors may turn to generic **div**s less often now that we have better semantic sectioning elements.

MARKUP TIP

It is possible to nest **div** elements within other **div** elements, but don't go overboard. Always strive to keep your markup as simple as possible, so add a **div** element only if it is necessary for logical structure, styling, or scripting. You want to avoid what developers call "divitis" or "div soup."

Define a Phrase with span

A **span** offers the same benefits as the **div** element, except it is used for phrase elements and does not introduce line breaks. Because **span**s are text-level elements, they may contain only text and other text-level elements (in other words, you cannot put headings, lists, content grouping elements, and so on in a **span**). Let's get right to some examples.

There is no **telephone** element, but we can use a **span** to give meaning to telephone numbers. In this example, each telephone number is marked up as a **span** and classified as "tel":

```
<ul>
  <li>John: <span class="tel">999.8282</span></li>
  <li>Paul: <span class="tel">888.4889</span></li>
  <li>George: <span class="tel">888.1628</span></li>
  <li>Ringo: <span class="tel">999.3220</span></li>
</ul>
```

You can see how the classified **span**s add meaning to what otherwise might be a random string of digits. As a bonus, the **span** element enables us to apply the same style to phone numbers throughout the site (for example, ensuring line breaks never happen within them by using a CSS **white-space: nowrap** declaration).

id and class Attributes

In the previous examples, we used the **id** and **class** attributes to provide context to generic **div** and **span** elements. **id** and **class** have different purposes, however, and it's important to know the difference.

Identification with id

The **id** attribute is used to assign a *unique* identifier to an element in the document. In other words, the value of **id** must be used only once in the document. This makes it useful for assigning a name to a particular element, as though it were a piece of data. See the sidebar **"id and class Values"** for information on providing values for the **id** attribute.

In this example, each book listing **div** is identified by the book's unique ISBN number (see **Note**) as the value of the **id** attribute. The values could also have been something simple (**product1**, **product2**, for example), as long as no two **id**s in the document had the same value.

```
<div id="ISBN0321127307">
  <img src="felici-cover.gif" alt="book cover">
  <p><cite>The Complete Manual of Typography</cite>, James Felici</p>
  <p>A combination of type history and examples of good and bad type.
  </p>
</div>

<div id="ISBN0881792063">
  <img src="bringhurst-cover.gif" alt="book cover">
```

id and class Values

The values for **id** and **class** attributes must contain at least one character (that is, they may not be empty), and **id** values may not contain any character spaces. You can use pretty much any character in the value.

Earlier versions of HTML had restrictions on **id** values (for example, they needed to start with a letter), but those restrictions were removed in HTML5.

NOTE

An ISBN (International Standard Book Number) is a unique number assigned to a published book or other piece of media. Look for one on the copyright page or in the bar code on the back cover.

```
  <p><cite>The Elements of Typographic Style</cite>, Robert Bringhurst
  </p>
  <p>This lovely, well-written book is concerned foremost with creating
beautiful typography.</p>
</div>
```

Web authors also use **id** when identifying the various sections of a page. In the following example, there may not be more than one element with the **id** of "links" or "news" in the document:

```
<section id="news">
  <!-- news items here -->
</section>

<aside id="links">
  <!-- list of links here -->
</aside>
```

Classification with class

> **MARKUP TIP**
>
> Use the **id** attribute to *identify*.
>
> Use the **class** attribute to *classify*.

The **class** attribute classifies elements into conceptual groups; therefore, unlike the **id** attribute, a **class** name may be shared by multiple elements. Making elements part of the same class enables you to easily apply styles to all of the labeled elements at once with a single style rule, or manipulate them all with a script. Let's start by classifying some elements in the earlier book example. In this first example, we've added **class** attributes to classify each **div** as a "listing" and to classify paragraphs as "descriptions":

```
<div id="ISBN0321127307" class="listing">
  <header>
  <img src="felici-cover.gif" alt="">
  <p><cite>The Complete Manual of Typography</cite>, James Felici</p>
  </header>
  <p class="description">A combination of type history and examples of
good and bad type.</p>
</div>

<div id="ISBN0881792063" class="listing">
  <header>
  <img src="bringhurst-cover.gif" alt="">
  <p><cite>The Elements of Typographic Style</cite>, Robert Bringhurst
  </p>
  </header>
  <p class="description">This lovely, well-written book is concerned
foremost with creating beautiful typography.</p>
</div>
```

Notice how the same element may have both a **class** and an **id**. It's also possible for elements to belong to multiple classes. When there is a list of **class** values, simply separate them with character spaces. In this example, we've classified each **div** as a "book" to set them apart from possible "audiobook" or "dvd" listings elsewhere in the document:

```
<div id="ISBN0321127307" class="listing book">
  <img src="felici-cover.gif" alt="CMT cover">
  <p><cite>The Complete Manual of Typography</cite>, James Felici</p>
  <p class="description">A combination of type history and examples of
```

```
good and bad type.</p>
</div>

<div id="ISBN0881792063" class="listing book">
  <img src="bringhurst-cover.gif" alt="ETS cover">
  <p><cite>The Elements of Typographic Style</cite>, Robert Bringhurst
  </p>
  <p class="description">This lovely, well-written book is concerned
  foremost with creating beautiful typography.</p>
</div>
```

Identify and Classify All the Elements

The **id** and **class** attributes are not limited to just **div** and **span**—they are two of the **global attributes** in HTML, which means you may use them with any HTML element (see **Note**). For example, you could give an ordered list an **id** of "directions" instead of wrapping it in a **div**:

```
<ol id="directions">
  <li>...</li>
  <li>...</li>
  <li>...</li>
</ol>
```

This section should have given you a good introduction to how to use the **class** and **id** attributes to add meaning and organization to documents. We'll work with them more in the style sheet chapters in **Part III, CSS for Presentation**.

NOTE

Other examples of global attributes are **style**, **tabindex**, *and* **title**. *See* **Appendix B, HTML Global Attributes**, *for a list of all the available global attributes with their values and definitions.*

IMPROVING ACCESSIBILITY WITH ARIA

As web designers, we must always consider the experience of users with assistive technologies for navigating pages and interacting with web applications. Your users may be listening to the content on the page read aloud by a screen reader or using keyboards, joysticks, voice commands, or other nonmouse input devices to navigate through the page.

Many HTML elements are plainly understood when you look at (or read) only the HTML source. Elements like the title, headings, lists, images, and tables have implicit meanings in the context of a page; however, generic elements like **div** and **span** lack the semantics necessary to be interpreted by an assistive device. In rich web applications, especially those that rely heavily on JavaScript, the markup alone does not provide enough clues as to how elements are being used or whether a form control is currently selected, required, or in some other state.

Fortunately, we have **ARIA** (**Accessible Rich Internet Applications**), a standardized set of attributes for making pages easier to navigate and interactive features easier to use. The specification was created and is maintained by a Working Group of the Web Accessibility Initiative (WAI), which is why you'll

also hear it referred to as WAI-ARIA. ARIA defines **roles**, **states**, and **properties** that developers can add to markup and scripts to provide richer semantic information. The HTML specification lists which roles and properties apply in the description of every element.

Roles

Roles describe an element's function or purpose in the context of the document. Examples include **`alert`**, **`button`**, **`dialog`**, **`slider`**, and **`menubar`**, to name only a few. For instance, as you saw earlier, you can turn an unordered list into a tabbed menu of options using style sheets, but what if you can't *see* that it is styled that way? Adding **`role="toolbar"`** to the list makes its purpose clear:

```
<ul id="tabs" role="toolbar">
  <li>A-G</li>
  <li>H-O</li>
  <li>P-T</li>
  <li>U-Z</li>
</ul>
```

Here's another example that reveals that the "status" **`div`** is used as an alert message:

```
<div id="status" role="alert">You are no longer connected to the
server.</div>
```

Some roles describe "landmarks" that help readers find their way through the document, such as **`navigation`**, **`banner`**, **`contentinfo`**, **`complementary`**, and **`main`**. You may notice that some of these sound similar to the page structuring elements that were added in HTML5, and that's no coincidence. One of the benefits of having improved semantic section elements is that they can be used as landmarks, so, for example, we can replace **`<div id="main" role="main">`** with **`<main>`**.

Most current browsers already recognize the implicit roles of the new elements, but some developers explicitly add ARIA roles until all browsers comply. The sectioning elements pair with the ARIA landmark roles in the following way:

```
<nav role="navigation">
<header role="banner"> (see Note)
<main role="main">
<aside role="complementary">
<footer role="contentinfo">
```

MARKUP TIP

It can be tempting to lean on ARIA rather than to focus on building a solid document structure through semantic markup, but that leads to many accessibility issues. Treat ARIA as a way of supplementing a solid foundation of semantic markup.

NOTE

The **`banner`** *role is used when the* **`header`** *applies to the whole page, not just a section or article.*

States and Properties

ARIA also defines a long list of states and properties that apply to interactive elements such as form widgets and dynamic content. States and properties are indicated with attributes prefixed with **`aria-`**, such as **`aria-disabled`**, **`aria-describedby`**, and many more.

The difference between a state and a property is subtle. For properties, the value of the attribute is more likely to be stable; examples include **`aria-labelledby`**, which associates labels with their respective form controls, and **`aria-haspopup`**, which indicates the element has a related pop-up menu. States have values that are more likely to be changed as the user interacts with the element, such as **`aria-selected`**.

CHARACTER ESCAPES

There's just one more text-related topic to cover before I close out this chapter. The title of this section makes it sound like someone left the gate open and all the characters got out. The real meaning is more mundane, albeit useful to know.

You already know that as a browser parses an HTML document, when it runs into a **<** symbol, it interprets it as the beginning of a tag. But what if you just need a less-than symbol in your text? Characters that might be misinterpreted as code need to be **escaped** in the source document. Escaping means that instead of typing in the character, you represent it by its numeric or named **character entity reference**. When the browser sees the character reference, it substitutes the proper character in that spot when the page is displayed.

There are two ways of referring to (escaping) a specific character:

- Using a predefined abbreviated name for the character (called a **named entity**). There are hundreds of named entities defined in HTML, and you can't make up your own.
- Using an assigned numeric value that corresponds to its position in a coded character set (**numeric entity**). Numeric values may be in decimal or hexadecimal format.

All character references begin with an **&** (ampersand) and end with a **;** (semicolon).

An example should make this clear. If, as mentioned previously, you want to use a less-than symbol in your text, you must use the named entity (**`<`**) or its numeric equivalent (**`<`**) where you want the symbol to appear (FIGURE 5-17). For example, you'd need to write:

```
<p>3 tsp. &lt; 3 Tsp.</p>
```

or:

```
<p>3 tsp. &#060; 3 Tsp.</p>
```

FURTHER READING

Obviously, this is not enough ARIA coaching to allow you to start confidently using it today, but it should give you a good feel for its value. When you are ready to dig in and take your skills to a professional level, here are some recommended resources:

The WAI-ARIA Working Draft (w3.org/TR/wai-aria-1.3/)

This is the current Working Draft of the specification as of this writing.

ARIA in HTML (w3.org/TR/html-aria/)

This W3C Working Draft helps developers use ARIA attributes with HTML correctly. It features a great list of every HTML element, whether it has an implicit role (in which case ARIA should *not* be used), and what roles, states, and properties apply.

ARIA resources at MDN Web Docs (developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA)

This site features lots of links to ARIA-related and up-to-date resources. It's a good starting point for exploration.

HTML5 Accessibility (html5accessibility.com)

This site tests which new HTML5 features are accessibly supported by major browsers.

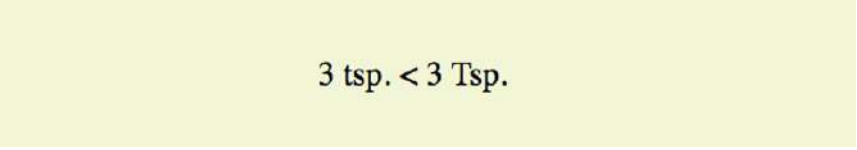


FIGURE 5-17. The special character is substituted for the character reference when the document is displayed in the browser.

When to Escape Characters

The majority of special characters do not need to be escaped if you are using the UTF-8 character encoding for your document (declared with a `meta` element in the `head` of the document). The browser will display most special characters in the source document just fine. However, there are a few instances in which you may need or want to use a character reference.

Characters from HMTL code syntax

■ MARKUP TIP

These character entities are useful when you need to show an example of HTML markup on a web page.

The <, >, &, ", and ' characters have special syntax meanings in HTML and may be misinterpreted as code. Therefore, the spec recommends that you escape <, >, and & characters in content. The most important is <, because it indicates the beginning of a tag. The ampersand (&) is generally safe unless the characters directly after it indicate a character reference.

If attribute values contain single or double quotes, escaping the quote characters in the values is advised. Quote marks are fine in the content and do not need to be escaped. TABLE 5-2 lists the entity names and decimal and hexadecimal numeric equivalents for each of these characters.

TABLE 5-2. Syntax characters and their character references

Character	Description	Entity name	Decimal no.	Hexadecimal no.
<	Less-than symbol	`<`	`<`	`<`
>	Greater-than symbol	`>`	`>`	`>`
"	Quotation mark	`"`	`"`	`"`
'	Apostrophe	`'`	`'`	`'`
&	Ampersand	`&`	`&`	`&`

Invisible or ambiguous characters

Some characters have no graphic display (see **Note**) and are difficult to see in the markup. These characters (listed in TABLE 5-3) include the non-breaking space (** **), which is used to ensure that a line doesn't break between two words. So, for instance, if I mark up my name like this:

```
Jennifer Robbins
```

I can be sure that my first and last names will always stay together on a line. Another use for non-breaking spaces is to separate digits in a long number, such as 32 000 000.

Zero-width spaces can be inserted in languages that do not use spaces between words to indicate where lines should break. A zero-width joiner is a non-printing space that causes neighboring characters to display in their connected forms (common in Arabic and Indic languages). Zero-width non-joiners prevent neighboring characters from joining to form ligatures or other connected forms.

NOTE

Some code editors do display invisible characters. For example, IntelliJ editors indicate non-breaking spaces with "NBSP".

TABLE 5-3. Invisible characters and their character references

Character	Description	Entity name	Decimal no.	Hexadecimal no.
(non-printing)	Non-breaking space			
(non-printing)	En space			
(non-printing)	Em space			
(non-printing)	Zero-width space	(none)		
(non-printing)	Zero-width non-joiner	‌	‌	‌
(non-printing)	Zero-width joiner	‍	‍	‍

Input limitations

If your keyboard or editing software does not include the character you need (or if you simply can't find it), you can use a character entity to make sure you get the character you want. The spec doesn't endorse this practice, so use the proper character in your source if you are able. TABLE 5-4 lists some special characters that may be less straightforward to type into the source

TABLE 5-4. Special characters and their character references

Character	Description	Entity name	Decimal no.	Hexadecimal no.
‘	Left curly single quote	‘	‘	‘
’	Right curly single quote	’	’	’
“	Left curly double quote	“	“	“
”	Right curly double quote	”	”	”
…	Horizontal ellipsis	…	…	…
©	Copyright	©	©	©
®	Registered trademark	®	®	®
™	Trademark	™	™	…
£	Pound	£	£	£
¥	Yen	¥	¥	¥
€	Euro	€	€	€
–	En dash	–	–	–
—	Em dash	—	—	—

A complete list of HTML named entities and their Unicode code points can be found as part of the HTML specification at ***html.spec.whatwg.org/multipage/named-characters.html#named-character-references***. For a more user-friendly listing of named and numerical entities, I recommend this archived page at the Web Standards Project: ***webstandards.org/learn/reference/charts/entities***.

PUTTING IT ALL TOGETHER

So far, you've learned how to mark up elements, and you've met all of the HTML elements for adding structure and meaning to text content. Now it's just a matter of practice. EXERCISE 5-3 gives you an opportunity to try out everything we've covered so far: document structure elements, grouping (block) elements, text-level elements, sectioning elements, and character entities.

EXERCISE 5-3. The Black Goose Bistro News page

Now that you've been introduced to all of the text elements, you can put them to work by marking up the News page for the Black Goose Bistro site. You can find the starter text and finished markup files at *learningwebdesign.com/6e/materials*. The resulting page is shown in FIGURE 5-18. Here's the text that we'll be marking up:

```
The Black Goose Bistro

News
Home
Menu
News
Contact

Summer Menu Items
posted by BGB, June 18, 2024
Our chef has been busy putting together the perfect menu for the summer
months. Stop by to try these appetizers and main courses while the days
are still long.

Appetizers
Black bean purses
Spicy black bean and a blend of Mexican cheeses wrapped in sheets of
phyllo and baked until golden. $3.95

Southwestern napoleons with lump crab -- new item!
Layers of light lump crab meat, bean and corn salsa, and our handmade
flour tortillas. $7.95

Main courses

Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then
grilled to perfection. Served with spicy peanut sauce and jasmine rice.
$12.95

Jerk rotisserie chicken with fried plantains -- new item!
Tender chicken slow-roasted on the rotisserie, flavored with spicy and
fragrant jerk sauce and served with fried plantains and fresh mango.
$12.95

Low and Slow
posted by BGB, November 15, 2024
<p>This week I am <em>extremely</em> excited about a new cooking
technique called <dfn><i>sous vide</i></dfn>. In <i>sous vide</i>
cooking, you submerge the food (usually vacuum-sealed in plastic) into a
water bath that is precisely set to the target temperature you want the
food to be cooked to. In his book, <cite>Cooking for Geeks</cite>, Jeff
Potter describes it as <q>ultra-low-temperature poaching.</q></p>

<p>Next month, we will be serving <b><i>Sous Vide</i> Salmon with Dill
Hollandaise</b>. To reserve a seat at the chef table, contact us before
<time datetime="20241130">November 30</time>.</p>

Location: Baker's Corner, Seekonk, MA
Hours: Tuesday to Saturday, 11am to 11pm

All content copyright 2024, Black Goose Bistro and Jennifer Robbins
```

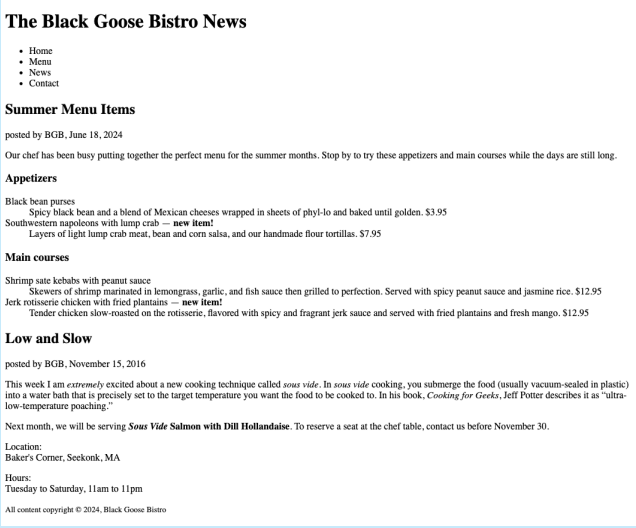
MARKUP TIPS

- Choose the element that best fits the meaning of the selected text.
- Don't forget to close elements with closing tags.
- Put all attribute values in quotation marks for clarity.
- Copy and paste is your friend when adding the same markup to multiple elements. Just be sure what you copied is correct before you paste it throughout the document.

NOTE

The "Low and Slow" paragraph is already marked up with the text-level elements from EXERCISE 5-2.

EXERCISE 5-3. Continued



The Black Goose Bistro News

- Home
- Menu
- News
- Contact

Summer Menu Items

posted by BGB, June 18, 2024

Our chef has been busy putting together the perfect menu for the summer months. Stop by to try these appetizers and main courses while the days are still long.

Appetizers

Black bean purses
Spicy black bean and a blend of Mexican cheeses wrapped in sheets of phyl-lo and baked until golden. $3.95
Southwestern napoleons with lump crab — **new item!**
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. $7.95

Main courses

Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice. $12.95
Jerk rotisserie chicken with fried plantains — **new item!**
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. $12.95

Low and Slow

posted by BGB, November 15, 2016

This week I am *extremely* excited about a new cooking technique called *sous vide*. In *sous vide* cooking, you submerge the food (usually vacuum-sealed in plastic) into a water bath that is precisely set to the target temperature you want the food to be cooked to. In his book, *Cooking for Geeks*, Jeff Potter describes it as "ultra-low-temperature poaching."

Next month, we will be serving ***Sous Vide* Salmon with Dill Hollandaise**. To reserve a seat at the chef table, contact us before November 30.

Location:
Baker's Corner, Seekonk, MA

Hours:
Tuesday to Saturday, 11am to 11pm

All content copyright © 2024, Black Goose Bistro

FIGURE 5-18. The finished menu page

1. Start by adding the DOCTYPE declaration to tell browsers this is a standards-compliant HTML document.
2. Add all the document structure elements first (`html`, `head`, `meta`, `title`, and `body`). Give the document the title "The Black Goose Bistro News."
3. Identify the top-level heading and the list of links as the header for the document by wrapping them in a `header` element (don't forget the closing tag). Within the header, the headline should be an `h1` and the list of links should be an unordered list (`ul`). Don't worry about making the list items links; we'll get to linking in the next chapter. Give the list more meaning by identifying it as the primary navigation for the site (`nav`).
4. The News page has two posts, titled "Summer Menu Items" and "Low and Slow." Mark up each one as an article.
5. Now, get the first article into shape. Let's create a header for this article that contains the heading (`h2` this time, because we've moved down in the document hierarchy) and the publication information (`p`). Identify the publication date for the article with the `time` element, just like in EXERCISE 5-2.
6. The content after the header is a simple paragraph. However, there are some interesting things going on in the menu. It's divided into two conceptual sections (Appetizers and Main Courses), so mark those up as `section` elements. Make sure that the final closing `</section>` tag appears before the closing `</article>` tag, so the elements are nested correctly and don't overlap. Finally, identify the sections with `id` attributes. Name the first one "appetizers" and the second "maincourses".
7. With our sections in place, now we can mark up the content. We're down to `h3` for the headings in each section. Choose the most appropriate list elements to describe the menu item names and their descriptions. Mark up the lists and each item within the lists.
8. Now we can add a few fine details. Classify each price as "price" using `span` elements.
9. Two of the dishes are new items. Change the double hyphens to an em dash character and mark up "new item!" as "strongly important." Classify the title of each new dish as "newitem" (use the existing `dt` element; there's no need to add a `span` this time). This allows us to target menu titles with the "newitem" class and style them differently than other menu items.
10. That takes care of the first article. The second article is already mostly marked up from the previous exercise, but you should mark up the header with the appropriate heading and publication date information.
11. So far, so good, right? Now make the remaining content that applies to the whole page a `footer`. Mark each line of content within the footer as a paragraph.
12. Next, give the location and hours information some context by putting them in a `div` named "about". Make the labels "Location:" and "Hours:" appear on lines by themselves by adding line breaks after them. Mark up the hours with the `time` element (you don't need the date or time zone portions).
13. Finally, copyright information is typically "small print" on a document, so mark it up accordingly. As the final touch, add a copyright symbol after the word "copyright" using the keyboard or the `©` character entity.

Save the document as *bistro_news.html*, and check your page in a browser. You can also use the Nu Html Checker (*validator.nu*) to make sure it's valid (a great way to spot mistakes). How did you do?

Want More Practice?

Try marking up your own résumé. Start with the raw text and add document structure elements, content grouping elements, and text-level elements, as you did in EXERCISE 5-3. If you don't see an element that matches your information just right, try using a `div` or a `span`.

TEST YOURSELF

Were you paying attention? Here is a set of questions to find out. You'll find the answers in **Appendix A**.

1. Add the markup to insert a thematic break between these paragraphs:

```
<p>People who know me know that I love to cook.</p>

<p>I've created this site to share some of my favorite recipes.</p>
```

2. What's the difference between a `blockquote` and a `q` element?

3. Which element displays whitespace exactly as it is typed into the source document?

4. What is the difference between a `ul` and an `ol` element?

5. How do you remove the bullets from an unordered list? (Be general, not specific.)

6. What element would you use to mark up "W3C" and provide its full name (World Wide Web Consortium)? Can you write out the complete markup?

7. What is the difference between `dl` and `dt`?

8. What is the difference between `id` and `class`?

9. What is the difference between an `article` and a `section`?

ELEMENT REVIEW: TEXT ELEMENTS

The global attributes apply to all text elements. Additional attributes are listed under their respective elements.

Page sections

`address`	Author contact information
`article`	Self-contained content
`aside`	Tangential content (sidebar)
`footer`	Related content
`header`	Introductory content
`nav`	Primary navigation
`section`	Conceptually related group of content

Heading content

`h1...h6`	Headings, levels 1 through 6

Grouping content elements and attributes

`blockquote`	Blockquote
`cite`	The URL of the cited content
`div`	Generic division
`figure`	Related image or resource
`figcaption`	Text description of a figure
`hr`	Paragraph-level thematic break (horizontal rule)
`main`	Primary content area of page or app
`p`	Paragraph
`pre`	Preformatted text

List elements and attributes

`dd`	Definition
`dl`	Definition list
`dt`	Term
`li`	List item (for **`ul`** and **`ol`**)
`value`	Provides a number for an **`li`** in an **`ol`**
`ol`	Ordered list
`reversed`	Numbers the list in reverse order
`start`	Provides the starting number for the list
`ul`	Unordered list

Breaks

`br`	Line break
`wbr`	Word break

Phrasing elements and attributes

`abbr`	Abbreviation
`b`	Added visual attention (bold)
`bdi`	Bidirectional isolation
`bdo`	Bidirectional override
`cite`	Citation
`code`	Code sample
`data`	Machine-readable equivalent
`del`	Deleted text
`cite`	The URL of cited content.
`datetime`	Specifies the date and time of a change
`dfn`	Defining term
`em`	Stress emphasis
`i`	Alternate voice (italic)
`ins`	Inserted text
`cite`	The URL of cited content
`datetime`	Specifies the date and time of a change
`kbd`	Keyboard input
`mark`	Highlighted text
`q`	Short inline quotation
`cite`	The URL of the cited content
`ruby`	Section containing ruby text
`rp`	Parentheses in ruby text
`rt`	Ruby annotation
`s`	Strike-through; incorrect text
`samp`	Sample output
`search`	Search- or filtering-related content
`small`	Annotation; "small print"
`span`	Generic phrase of text
`strong`	Strong importance
`sub`	Subscript
`sup`	Superscript
`time`	Machine-readable time data
`datetime`	Provides machine readable date/time
`pubdate`	Indicates the time refers to publication
`u`	Added attention (underline)

CHAPTER 6

ADDING LINKS

IN THIS CHAPTER

Linking to external pages

Linking to documents on your own server

Linking to a specific point in a page

Targeting new windows

Mail and telephone links

If you're creating a page for the web, chances are you'll want to link to other web pages and resources, whether on your own site or someone else's. Linking, after all, is what the web is all about. In this chapter, we'll look at the markup that makes links work—links to other sites, to your own site, and within a page. There is one element that makes linking possible: the **anchor** (**a**).

`<a>...</a>`

Anchor element (hypertext link)

To make a selection of text a link, wrap it in opening and closing **<a>** tags and use the **href** attribute to provide the URL of the target page. The content of the anchor element becomes the hypertext link. Here is an example that creates a link to the O'Reilly Media site:

```
<a href="https://www.oreilly.com">Go to the O'Reilly Media site</a>
```

To make an image a link, wrap anchor tags around the **img** element:

```
<a href="https://www.oreilly.com"><img src="tarsierlogo.gif"
alt="O'Reilly Media site"></a>
```

By the way, you can put any HTML content element in an anchor to make it a link, not just images. As of HTML5, it is valid to put multiple elements in a single **a** element.

Nearly all graphical browsers display linked text as blue and underlined by default. Visited links generally display in purple. Users can change these colors in their browser preferences, and, of course, you can change the appearance of links for your sites using style sheets. I'll show you how in **Chapter 13, Colors and Backgrounds**.

When a user clicks or taps the linked text or image, the page specified in the anchor element loads in the browser window. FIGURE 6-1 shows how this works, using the example given previously.

■ AT A GLANCE

Anchor Structure

The simplified structure of the anchor element is as follows:

```
<a href="url">linked content</a>
```

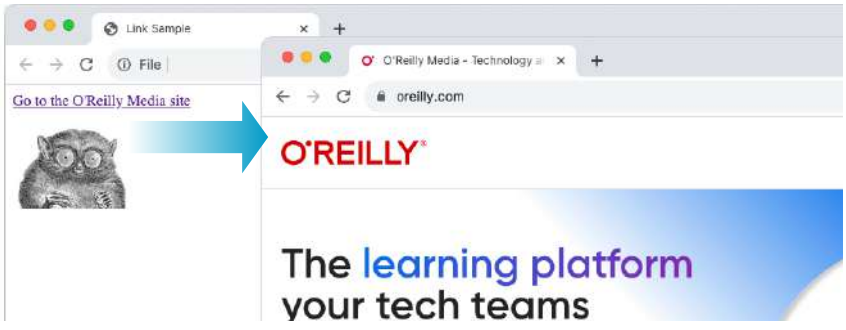


FIGURE 6-1. When a user clicks or taps the linked text or image, the page specified in the anchor element loads in the browser window.

THE HREF ATTRIBUTE

The **`href`** (hypertext reference) attribute provides the browser with the address (URL) of the page or resource that you want to link to. The URL must always appear in quotation marks. Most of the time you'll point to other HTML documents; however, you can also point to other web resources, such as images, audio files, and video files.

Because there's not much to slapping anchor tags around some content, the real trick to linking comes in getting the URL correct. There are two ways to specify the URL:

Absolute URLs

Absolute URLs provide the full URL for the document, including the protocol (**`https://`** or **`http://`**):

```
href="https://www.oreilly.com/"
```

You need to use an absolute URL when pointing to a document out on the web (i.e., not on your own server).

Sometimes, when the page you're linking to has a long URL pathname, the link can end up looking pretty confusing (FIGURE 6-2). Just keep in mind that the structure is still a simple container element with one attribute. Don't let the long pathname intimidate you.

Relative URLs

Relative URLs describe the pathname to a file *relative* to the current document. Relative URLs can be used when you are linking to another document on your own site (i.e., on the same server). They don't require the protocol or domain name, just the pathname:

```
href="recipes/index.html"
```

In this chapter, we'll add links using absolute and relative URLs for the Jen's Kitchen website (see FIGURE 6-3). Absolute URLs are easy, so let's get them out of the way first.

■ MARKUP TIP

URL Wrangling

If you're linking to a page with a long URL, it's helpful to copy the URL from the location toolbar in your browser and paste it into your document. That way, you avoid the risk of mistyping a single character and breaking the whole link.

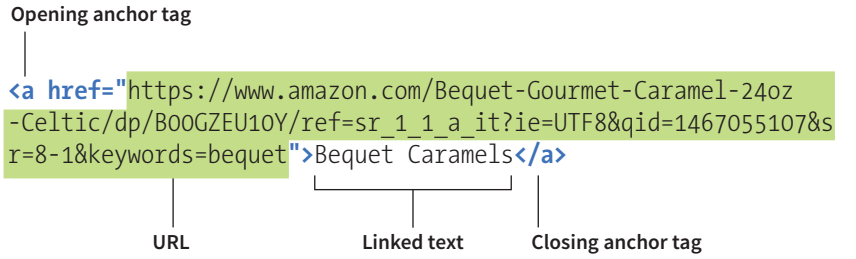


FIGURE 6-2. An example of a long URL—although it may make the anchor tag look confusing, the structure is the same

LINKING TO PAGES ON THE WEB

Many times, you'll want to create a link to a page that you've found on the web. This is known as an **external link** because it is going to a page outside of your own server or site. To make an external link, provide the absolute URL, beginning with **`https://`** (the protocol). This tells the browser, "Go out on the web and get the following document."

Let's add some external links to the Jen's Kitchen home page (FIGURE 6-3). First, I'm going to link the list item "The Food Network" to *foodnetwork.com*. I marked up the link text in an anchor element by adding opening and closing anchor tags:

```
<li><a>The Food Network</a></li>
```

Notice that I've added the anchor tags *inside* the list item (**`li`**) element. That's because only **`li`** elements are permitted to be children of a **`ul`** element; placing an **`a`** element directly inside the **`ul`** element would be invalid HTML. Next, I'll add the **`href`** attribute with the complete URL for the site:

```
<li><a href="https://www.foodnetwork.com">The Food Network</a></li>
```

And *voilà*! Now "The Food Network" appears as a link and takes our visitors to that site when they click or tap it. Give it a try in EXERCISE 6-1.

■ TRY IT

Working Along with Jen's Kitchen

All the files for the Jen's Kitchen website are available online at *learningwebdesign.com/6e/materials*. Download the entire directory, making sure not to change the way its contents are organized. The pages aren't much to look at, but they will give you a chance to develop your linking skills.

The resulting markup for all of the exercises is also provided.

FIGURE 6-3. The Jen's Kitchen page

EXERCISE 6-1. Make an external link

Open the file *index.html* in the *jenskitchen* folder. Make the list item "Epicurious" link to the web page at *www.epicurious.com*, following the Food Network link example:

```
<ul>
  <li><a href="https://www.foodnetwork.com/">The Food Network</a></li>
  <li>Epicurious</li>
</ul>
```

When you are done, save *index.html* and open it in a browser. If you have an internet connection, you can click your new link and go to the Epicurious site. If the link doesn't take you there, go back and make sure that you didn't miss anything in the markup.

LINKING WITHIN YOUR OWN SITE

A relative pathname describes how to get to the linked document, starting from the location of the current document.

A large portion of the linking you'll do is between pages of your own site: from the home page to section pages, from section pages to content pages, and so on. In these cases, you can use a relative URL—one that calls for a page on your own server.

If the protocol (`https://`) is not included in the URL, the browser looks on the current server for the linked document. A **pathname**, the notation used to point to a particular file or directory (see **Note**), tells the browser where to find the file. Web pathnames follow the Unix convention of separating directory and filenames with forward slashes (/). A relative pathname describes how to get to the linked document, starting from the location of the current document.

Relative pathnames can get a bit tricky. In my teaching experience, nothing trips up beginners like writing relative pathnames, so we'll take it one step at a time. I recommend you do EXERCISES 6-2 through 6-8 as we go along.

All of the pathname examples in this section are based on the structure of the Jen's Kitchen site shown in FIGURE 6-4. When you diagram the structure of the directories for a site, it generally ends up looking like an inverted tree with the root directory at the top of the hierarchy. All sites have a **root directory**, the directory that contains all the directories and files for the site. For the Jen's Kitchen site, the root directory is named *jenskitchen*. For another way to look at it, there is also a view of the directory and subdirectories as they appear in the Finder on a Mac in the bottom-left corner of FIGURE 6-4.

NOTE

On PCs and Macs, files are organized into "folders," but in the web development world, it is more common to refer to the equivalent and more technical term "directory." A folder is just a directory with a cute icon.

Important Pathname Don'ts

When writing relative pathnames, follow these rules to avoid common errors:

- Don't use backslashes (\). Web URL pathnames use forward slashes (/) only.
- Don't start with the drive name (`D:`, `C:`, etc.). Although your pages will link to each other successfully while they are on your own computer, once they are uploaded to the web server, the drive name is irrelevant and will break your links.
- Don't start with `file://`. This also indicates that the file is local and causes the link to break when the HTML document is uploaded to the server.

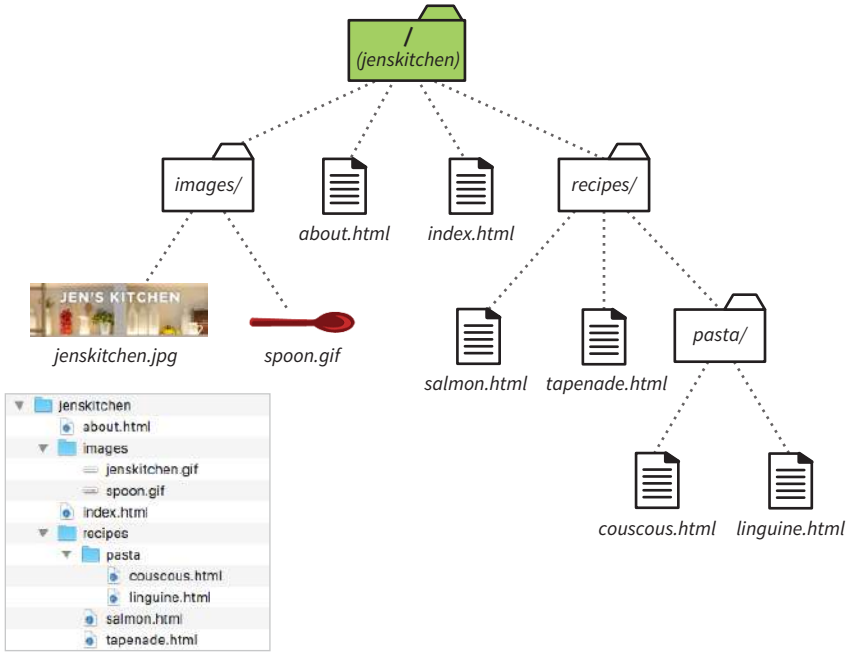


FIGURE 6-4. A diagram of the *jenskitchen* site structure

Linking Within a Directory

The most straightforward relative URL points to another file within the same directory. When linking to a file in the same directory, you need to provide only the name of the file (its **filename**). When the URL is just a filename, the server looks in the current directory (that is, the directory that contains the document with the link) for the file.

In this example, I want to make a link from the home page (*index.html*) to a general information page (*about.html*). Both files are in the same directory (*jenskitchen*). So, from the home page, I can make a link to the information page by simply providing its filename in the URL (FIGURE 6-5):

```
<a href="about.html"><p>About this site</></a>
```

EXERCISE 6-2 gives you a chance to mark up a simple link yourself.

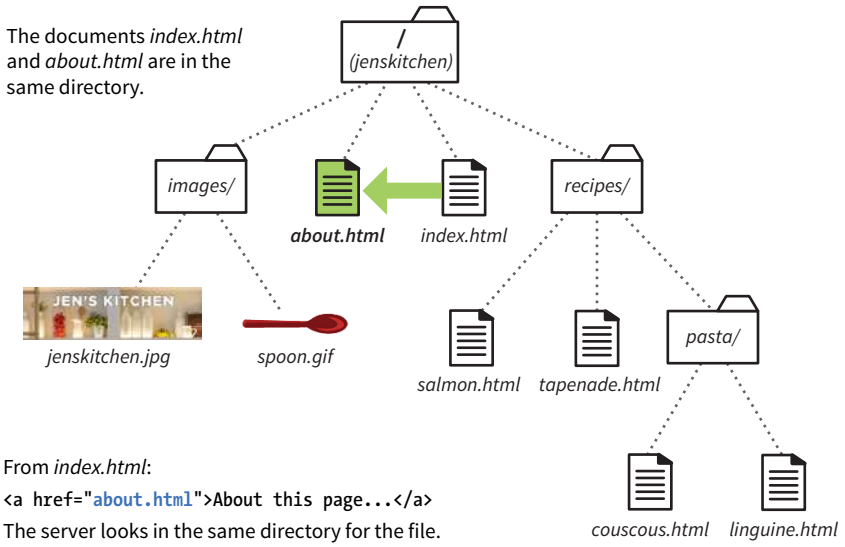


FIGURE 6-5. Writing a relative URL to another document in the same directory

Servers look in the same directory as the current document when the URL is just a filename.

EXERCISE 6-2. Link in the same directory

Open the file *about.html* in the *jenskitchen* folder. Make the paragraph "Back to the home page" at the bottom of the page link back to *index.html*. Because anchor elements may contain block elements, you can put the anchor tags around the p element:

```
<p>Back to the home page</p>
```

When you are done, save *about.html* and open it in a browser. You don't need an internet connection to test links locally (that is, on your own computer). Clicking the link should take you back to the home page.

Linking to a Lower Directory

But what if the files aren't in the same directory? You have to give the browser directions by including the pathname in the URL. Let's see how this works.

Getting back to the example, the recipe files are stored in a subdirectory called *recipes*. I want to make a link from *index.html* to a file in the *recipes* directory called *salmon.html*. The pathname in the URL tells the browser to look for a directory called *recipes* in the current directory first, and then look in the *recipes* directory for the file *salmon.html* (FIGURE 6-6):

```
<li><a href="recipes/salmon.html">Garlic Salmon</a></li>
```

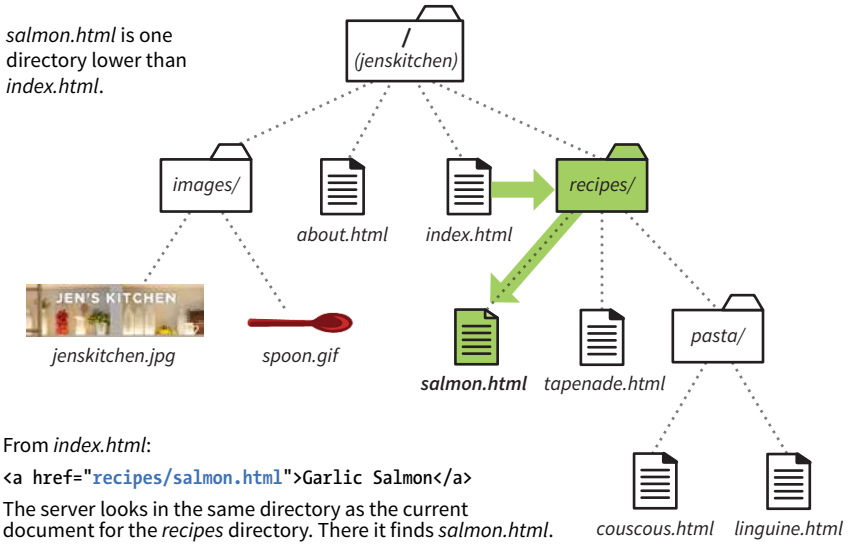


FIGURE 6-6. Writing a relative URL to a document that is one directory level lower than the current document

Have a try at linking to a file in a directory in EXERCISE 6-3.

EXERCISE 6-3. Link to a file in a directory

Open the file *index.html* in the *jenskitchen* folder. Make the list item "Tapenade (Olive Spread)" link to the file *tapenade.html* in the *recipes* directory. Remember to nest the elements correctly:

```
<li>Tapenade (Olive Spread)</li>
```

When you are done, save *index.html* and open it in a browser. You should be able to click your new link and see the recipe page for tapenade. If not, make sure that your markup is correct and that the directory structure for *jenskitchen* matches the examples.

Now let's link down to the file called *couscous.html*, which is located in the *pasta* subdirectory. All we need to do is provide the directions through two subdirectories (*recipes*, then *pasta*) to *couscous.html* (FIGURE 6-7):

```
<li><a href="recipes/pasta/couscous.html">Couscous...</a></li>
```

Directories are separated by forward slashes. The resulting anchor tag tells the browser, "Look in the current directory for a directory called *recipes*. There you'll find a directory called *pasta*, and in there is the file *couscous.html*."

Now that we've done two directory levels, you should get the idea of how pathnames are assembled. This same method applies for relative pathnames that drill down through any number of directories. Just start with the name of the directory that is in the same location as the current file, and follow each directory name with a slash until you get to the linked filename. You can try this yourself in EXERCISE 6-4.

When you link to a file in a lower directory, the pathname contains the names of each subdirectory you go through to get to the file.

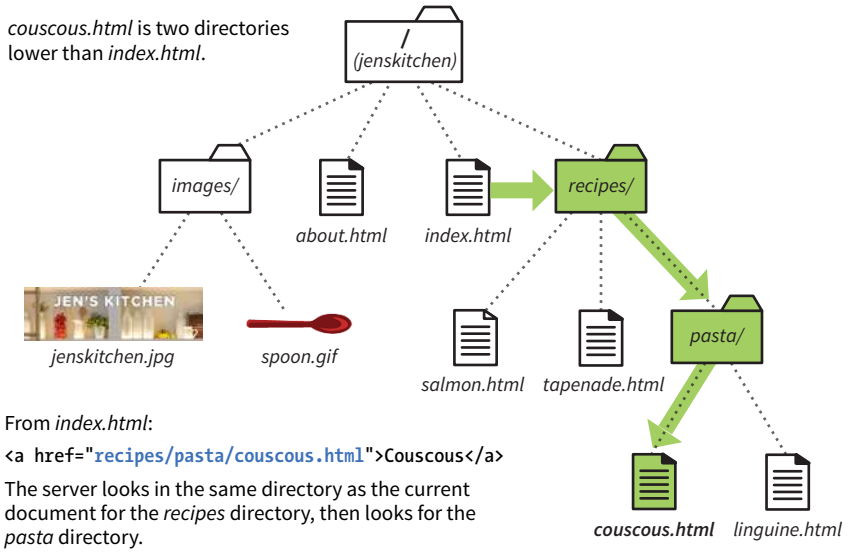


FIGURE 6-7. Writing a relative URL to a document that is two directory levels lower than the current document

EXERCISE 6-4. Link two directories down

Open the file *index.html* in the *jenskitchen* folder. Make the list item "Linguine with Clam Sauce" link to the file *linguine.html* in the *pasta* directory:

```
<li>Linguine with Clam Sauce</li>
```

When you are done, save *index.html* and open it in a browser. Click the new link to get the delicious recipe.

Linking to a Higher Directory

Each ../ at the beginning of the pathname tells the browser to go up one directory level to look for the file.

So far, so good, right? Now it gets more interesting. This time we're going to go in the other direction and make a link from the salmon recipe page back to the home page, which is one directory level up.

In Unix, there is a pathname convention just for this purpose: the "dot-dot-slash" (**../**). When you begin a pathname with **../**, it's the same as telling the browser "back up one directory level, and then follow the path to the specified file." If you're familiar with browsing files on your desktop, it's helpful to know that a **../** has the same effect as clicking the Up button in Windows Explorer or the left-arrow button in the Finder on macOS.

Let's start by making a link from *salmon.html* back to the home page (*index.html*). Because *salmon.html* is in the *recipes* subdirectory, we need to go back up to the *jenskitchen* directory to find *index.html*. This pathname tells the browser to back up one level, then look in that directory for *index.html* (FIGURE 6-8):

```
<p><a href="../index.html">[Back to home page]</a></p>
```

Note that the **../** stands in for the name of the higher directory, so we don't need to write out *jenskitchen* in the pathname.

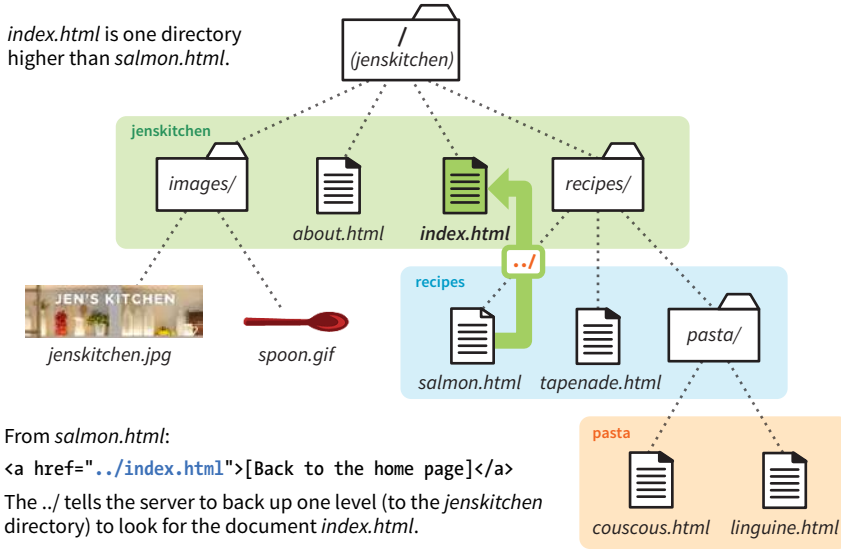


FIGURE 6-8. Writing a relative URL to a document that is one directory level higher than the current document

Try adding a dot-dot-slash pathname to a higher directory in EXERCISE 6-5.

But how about linking back to the home page from *couscous.html*? Can you guess how you'd back your way out of two directory levels? Simple: use the dot-dot-slash twice. A link on the *couscous.html* page back to the home page (*index.html*) would look like this:

```
<p><a href="../../index.html">[Back to home page]</a></p>
```

The first **../** backs up to the *recipes* directory, and the second **../** backs up to the top-level directory (*jenskitchen*), where *index.html* can be found (FIGURE 6-9). Again, there is no need to write out the directory names; the **../** does it all.

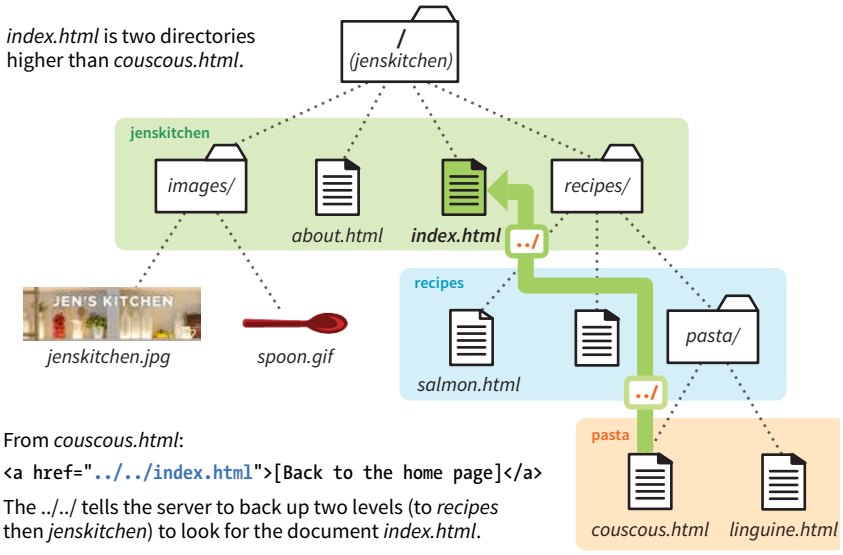


FIGURE 6-9. Writing a relative URL to a document that is two directory levels higher than the current document

You can try this out in EXERCISE 6-6.

NOTE

I confess to still sometimes silently chanting "go-up-a-level, go-up-a-level" for each **../** *when trying to decipher a complicated relative URL. It helps me sort things out.*

EXERCISE 6-5. Link to a higher directory

Open the file *tapenade.html* from the *recipes* directory. At the bottom of the page, you'll find this paragraph:

```
<p>[Back to the home page]</p>
```

Using the notation described in this section, make this text link back to the home page (*index.html*), located one directory level up.

EXERCISE 6-6. Link up two directory levels

Open the file *linguine.html* and make the last paragraph link back to the home page by using **../../**:

```
<p>[Back to the home page]</p>
```

When you are done, save the file and open it in a browser. Clicking or tapping the link should take you to the home page.

Linking with Site Root Relative Pathnames

So far, all of the pathnames we've looked at have been relative to the document with the link. Another way to write a relative pathname is to start at the site's root directory and list each subdirectory you have to go through to get to the file you want to link to. This type of pathname is known as **site root relative**. Site root relative links are generally preferred because of their flexibility.

A forward slash (/) at the start of a pathname indicates that the path begins at the root directory.

In the Unix pathname convention, a forward slash (/) at the start of the pathname indicates that the path begins at the root directory. The site root relative pathname in the following link tells the browser, "starting at the top-level directory for this site, open the *recipes* directory, and then find the *salmon.html* file" (FIGURE 6-10):

```
<a href="/recipes/salmon.html">Garlic Salmon</a>
```

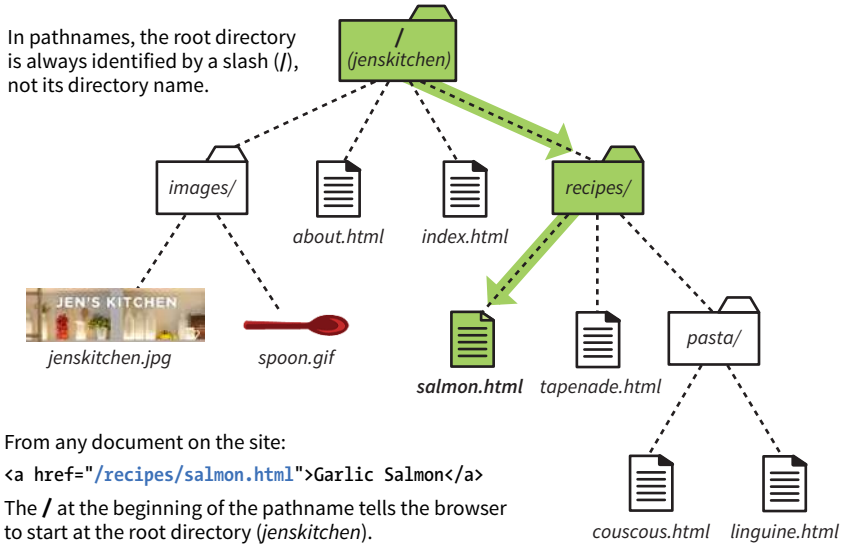


FIGURE 6-10. Writing a relative URL starting at the root directory

Note that you don't need to (and you shouldn't) write the name of the root directory (*jenskitchen*) in the path—the forward slash (/) at the beginning represents the top-level directory in the pathname. From there, just specify the directories the browser should look in.

Because this type of link starts at the root to describe the pathname, it works from any document on the server, regardless of which subdirectory it may be located in. Site root relative links are useful for content that might not always be in the same directory, or for dynamically generated material. They also make it easy to copy and paste links between documents. On the downside, however, they won't work on your local machine, because they will be relative to your hard drive. See the **"Local Site Root Relative Pathnames"** sidebar for a workaround.

Local Site Root Relative Pathnames

One option to be able to open site root relative links locally is to install server software on your machine. Another popular option is to use some sort of build tool (such as Node.js) to help build the site. Many build tools create a local server on your machine, so site root relative pathnames will work fine. If neither installing server software nor using an advanced build tool is appropriate for you (if you are just starting out, they may not be), then it's probably worth avoiding site root relative pathnames to simplify your testing process while you're learning.

Writing Pathnames to Images

The **src** attribute in the **img** element works the same as the **href** attribute in anchors. Because you'll most likely be using images from your own server, you'll generally provide relative URLs for **src**.

Let's look at a few examples from the Jen's Kitchen site. First, to add an image to the *index.html* page, you'd use the following markup:

```
<img src="images/jenskitchen.gif" alt="Jen's Kitchen">
```

This URL says, "Look in the current directory (*jenskitchen*) for the *images* directory; in there you will find *jenskitchen.gif*."

Now for the *pièce de resistance*...adding an image to the file *couscous.html*:

```
<img src="../../images/spoon.gif" alt="spoon illustration">
```

This is a little more complicated than what we've seen so far. This pathname tells the browser to go up two directory levels to the top-level directory and, once there, look in the *images* directory for an image called *spoon.gif*. Whew!

Of course, you could simplify that path by going the site root relative route, in which case the pathname to *spoon.gif* (and any other file in the *images* directory) could be accessed like this:

```
<img src="/images/spoon.gif" alt="spoon illustration">
```

The trade-off is that you may not see the image in place until the site is uploaded to the server, but it does make maintenance easier once it's there.

EXERCISE 6-7 will give you a chance to put relative pathnames to use.

EXERCISE 6-7. Try a few more

Before we move on, why not try your hand at writing a few more relative URLs to make sure you've really gotten it? You can write your answers on the page, or, if you want to test your markup to see whether it works, make changes in the actual files. Note that the text shown here isn't included on the exercise pages—you'll need to add it before you can create the links (for example, you'll need to type in "Go to the Tapenade recipe" to apply the first link). The final code is in the finished exercise files in the *materials* folder.

1. Create a link on *salmon.html* to *tapenade.html*:

   ```
   Go to the Tapenade recipe
   ```

2. Create a link on *couscous.html* to *salmon.html*:

   ```
   Try this with Garlic Salmon.
   ```

3. Create a link on *tapenade.html* to *linguine.html*:

   ```
   Try the Linguine with Clam Sauce
   ```

4. Create a link on *linguine.html* to *about.html:*

   ```
   About Jen's Kitchen
   ```

5. Create a link on *tapenade.html* to ***www.allrecipes.com***:

   ```
   Go to Allrecipes.com
   ```

NOTE

Most of the pathnames in EXERCISE 6-7 *could be site root relative, but write them relative to the listed document for the practice.*

Linking to a Specific Point in a Page

Did you know you can link to a specific point in a web page? This is useful for providing shortcuts to information at the bottom of a long, scrolling page or for getting back to the top of a page with just one click or tap. Linking to a specific point in the page is known as linking to a document **fragment**.

NOTE

Linking to another spot on the same page works well for long, scrolling pages, but the effect may be lost on a short web page.

Linking to a fragment is a two-part process: first identify the destination in the page, then make a link to it. The following example shows how to create an alphabetical index at the top of a glossary page that links down to a particular section (FIGURE 6-11). We'll create a link from the letter H, so that when users click that letter, they'll jump to the "H" heading lower on the page.

Step 1: Identifying the destination

This step is like planting a flag in the document that you can get back to easily. To create a destination, use the **id** attribute to give the target element in the document a unique name (that's "unique" as in the name may appear only once in the document, not "unique" as in funky and interesting). In web lingo, this is the **fragment identifier**.

You may remember the **id** attribute from **Chapter 5, Marking Up Text**, where we used it to name generic **div** and **span** elements. Here, we're going to use it to name an element so that it can serve as a fragment identifier—that is, the destination of a link.

Here is a sample of the source for the glossary page. Because we want users to be able to link directly to the "H" heading, we'll add the **id** attribute to it and give it the value **startH** (FIGURE 6-11 [**1**]):

```
<h2 id="startH">H</h2>
```

Step 2: Linking to the destination

Fragment names in links must be preceded by an octothorpe symbol (#).

With the identifier in place, now we can make a link to it. At the top of the page, we'll create a link down to the **startH** fragment [**2**]. As for any link, we use the **a** element with the **href** attribute to provide the location of the link. To indicate that we're linking to a fragment, we use the octothorpe symbol (**#**), also called a hash, pound, or number symbol, before the fragment name:

```
<p>... F | G | <a href="#startH">H</a> | I | J ...</p>
```

And that's it. Now when someone clicks or taps the H from the listing at the top of the page, the browser will jump down and display the section starting with the "H" heading [**3**].

1. Identify the destination by using the **id** attribute.

```
<h2 id="startH">H</h2>
<dl>
<dt>hexadecimal</dt>
...
```

2. Create a link to the destination. The # before the name is necessary to identify this as a fragment and not a filename.

```
<p>... | F | G | <a href="#startH">H</a> | I | J ...</p>
```

3.

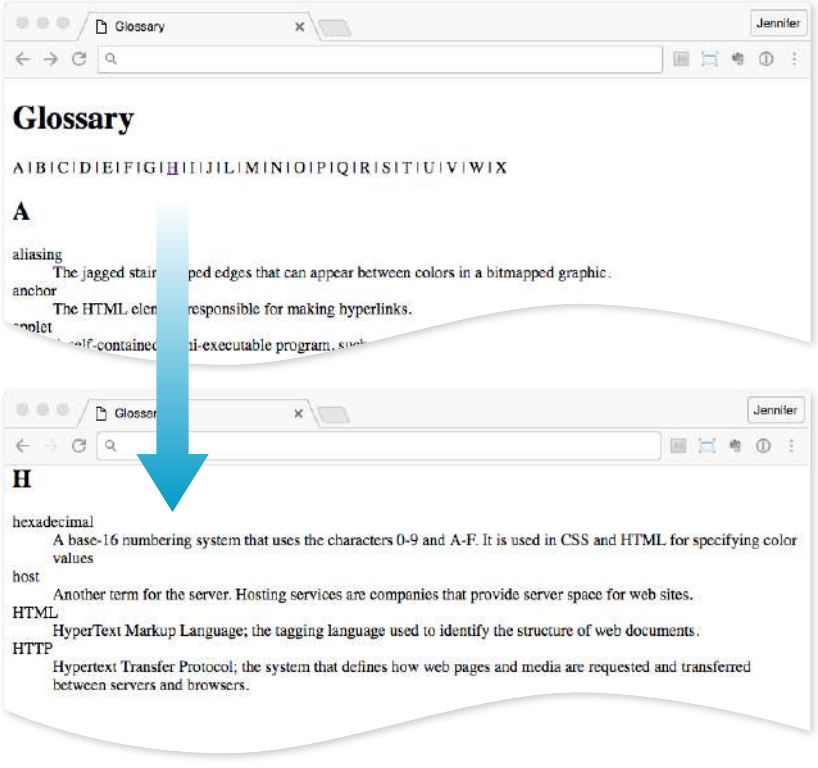

Glossary
Jennifer
Glossary
A|B|C|D|E|F|G|H|I|J|L|M|N|O|P|Q|R|S|T|U|V|W|X
A
aliasing
The jagged stair ped edges that can appear between colors in a bitmapped graphic.
anchor
The HTML elen responsible for making hyperlinks.
H
hexadecimal
A base-16 numbering system that uses the characters 0-9 and A-F. It is used in CSS and HTML for specifying color values
host
Another term for the server. Hosting services are companies that provide server space for web sites.
HTML
HyperText Markup Language; the tagging language used to identify the structure of web documents.
HTTP
Hypertext Transfer Protocol; the system that defines how web pages and media are requested and transferred between servers and browsers.

FIGURE 6-11. Linking to a specific destination (a fragment) within a single web page

Linking to a Fragment in Another Document

You can link to a fragment in another document by adding the fragment name to the end of the URL (absolute or relative). For example, to make a link to the "H" heading of the glossary page from another document in that directory, the URL would look like this:

```
<a href="glossary.html#startH">See the Glossary, letter H</a>
```

You can even link to specific destinations in pages on other sites by putting the fragment identifier at the end of an absolute URL, like so:

```
<a href="https://www.example.com/glossary.html#startH">See the
Glossary, letter H</a>
```

USABILITY TIP

To the Top!

It's common practice to add a link back up to the top of the page when linking into a long page of text. This alleviates the need for users to scroll back after every link. To do that, name an element at the top of the page with **id="top"**, such as a **header**, **h1**, **div**, or even the **body** element. At the bottom of the page, or at points throughout, add a link (for example, **Back to the top**) to the fragment identifier you set higher up.

EXERCISE 6-8. Linking to a fragment

Want some practice linking to specific destinations? Open *glossary.html* in the *materials* folder for this chapter. It looks just like the document in FIGURE 6-11.

1. Identify the **h2** "A" as a destination for a link by naming it "startA" with an **id** attribute:

   ```
   <h2 id="startA">A</h2>
   ```

2. Make the letter A at the top of the page a link to the identified fragment. Don't forget the #:

   ```
   <a href="#startA">A</a>
   ```

   Repeat steps 1 and 2 for every letter across the top of the page until you really know what you're doing (or until you get bored).

3. You can help users get back to the top of the page, too. Make the heading "Glossary" a fragment identifier named "top":

   ```
   <h1 id="top">Glossary</h1>
   ```

4. Add a paragraph element containing "TOP" at the end of one of the lettered sections. Make "TOP" a link to the identifier that you just made at the top of the page:

   ```
   <p><a href="#top">TOP</a></p>
   ```

   Copy this code and paste it at the end of every letter section. Now your readers can get back to the top of the page easily throughout the document.

Of course, you don't have any control over the named fragments in other people's web pages. The destination points must be inserted by the author of those documents in order for them to be available to you (some developers help their brothers and sisters out by proactively adding **id**s at the beginning of any thematic section of content). The only way to know whether they are there and where they are is to view the source for the page (as described in EXERCISE 2-1) and look for them in the markup. If the fragments in external documents move or go away, the page will still load; the browser will just go to the top of the page, as it does for regular links.

EXERCISE 6-8 gives you an opportunity to add links to some additional fragments in the example glossary page.

TARGETING A NEW BROWSER WINDOW

One problem with putting links on your page is that when people click them, they may never come back to your content. The traditional solution to this dilemma has been to make the linked page open in a new browser window (or tab). That way, your visitors can check out the link and still have your content available where they left it.

Be aware that opening new browser windows can cause hiccups in the user experience of your site. Opening new windows is problematic for accessibility, and may be confusing to some users. They might not be able to tell that a new window has opened, or they may never find their way back to the original page. At the very least, new windows may be perceived as an annoyance rather than a convenience.

Opening links in new tabs or windows can also bring performance and security issues. In some browsers, if the link is opened in a new window, it will still be sharing resources with your site. In other words, if the linked-to site is heavy and slow, it can slow down the performance of your site as well. By default, in older browsers, the new page can also access your page with some clever JavaScript, leading to security issues. I'll discuss how you can protect yourself from both the performance and security risks momentarily, but the upshot is that you should consider carefully whether you need a new window and whether the benefits outweigh the potential drawbacks.

To open a new browser window with markup, use the **target** attribute in the anchor (**a**) element to tell the browser the name of the window in which you want the linked document to open. You can set the value of **target** to **_blank** or to any name of your choosing. Remember that with this method, the document will generally open in a new tab or in a new window the same size as the most recently opened window in the user's browser. The new window may or may not be brought to the front, depending on the browser and device used.

Setting **target="_blank"**, as is done here, always causes the browser to open a fresh window:

```
<a href="https://www.oreilly.com" target="_blank">O'Reilly</a>
```

There's nothing wrong with this, per se, as long as it is not overused. Just keep in mind that if you include **target="_blank"** for every link, every link will launch a new window, potentially leaving your user with a mess of open windows.

Another method is to give the target window a specific name, which can then be used by subsequent links. You can give the window any name you like ("new," "sample," whatever), as long as it doesn't start with an underscore. The following link will open a new window called "display":

```
<a href="https://www.oreilly.com" target="display">O'Reilly</a>
```

If you target the "display" window from every other link on the page, each linked document will open in the same second window. Unfortunately, if that second window stays hidden behind the user's current window, it may look as though the links simply don't work.

With either approach, you'll want to make sure to include the **rel** attribute and set its value to **noOpener**. This tells the browser to make sure that the newly opened page isn't able to access your page, avoiding the potential the security and performance issues I mentioned earlier. It's definitely worth noting that current versions of all major browsers now implicitly provide this same level of protection, regardless of whether you set the value or not. However, older versions, as well as the current version of Android WebView (the browser used by Android apps), still have the vulnerability, so for now, setting **rel="noOpener"** as shown here is highly recommended:

```
<a href="https://www.oreilly.com" target="display"
rel="noOpener">O'Reilly</a>
```

You can decide which method (a new window for every link or reusing named windows) is most appropriate for your content and interface.

MAIL LINKS

Here's a nifty little linking trick: by using the **mailto:** protocol in a link, you can link to an email address. When the user clicks a **mailto:** link, the browser opens a new mail message preaddressed to that address in a designated mail program (see the **"Spam-Bots"** sidebar for a warning before including your own work or personal email address in such a link, however).

A sample **mailto:** link is shown here:

```
<a href="mailto:alklecker@example.com">alklecker@example.com</a>
```

As you can see, it's a standard anchor element with the **href** attribute, but the value is set to **mailto:***name@address.com*.

Spam-Bots

Putting an email address in your document source makes it susceptible to receiving unsolicited junk email (known as **spam**). People who generate spam lists sometimes use automated search programs (called **bots**) to scour the web for email addresses.

If you want your email address to display on the page so that humans can identify it but robots can't, you can deconstruct the address in a way that is still understandable to people—for example, "you [-at-] example [dot] com."

That trick won't work in a **mailto:** link, though, because the actual email address must be provided as an attribute value. One solution is to encrypt the email address by using JavaScript. If you're up for trying your hand at some JavaScript, there's an easy-to-follow tutorial on hiding email addresses on CSS-Tricks (*css-tricks.com/how-to-safely-share-your-email-address-on-a-website*).

If JavaScript isn't an option and you don't want to risk getting spammed, keep your email address out of your HTML documents. Using a contact form is a good alternative (web forms are coming up in **Chapter 9, Forms**).

You can also prefill the email subject line by adding **?subject=*<subject>*** to the **mailto:** link:

```
<a href="mailto:jen@example.com?subject=Reader Feedback">jen@example.
com</a>
```

WARNING

Be aware that browsers are increasingly blocking **mailto:** *links, so they will not work for everyone.*

Be aware that the user's browser must be configured to launch a mail program, so the effect won't work for users with web-based email accounts such as Gmail. If you use the email address itself as the linked text, as in the previous example, nobody will be left out if the **mailto:** function doesn't work (a nice little example of progressive enhancement).

TELEPHONE LINKS

Keep in mind that the smartphones many people are using to access your site can also be used to make phone calls! Why not save your visitors a step by letting them dial a phone number on your site simply by tapping on it on the page? The syntax uses the **tel:** protocol and is very simple:

```
<a href="tel:+1-800-555-1212">Call us at (800) 555-1212</a>
```

When mobile users tap the link, what happens depends on the device: Android launches the phone app, and iOS launches a dialog box giving the user the option to call, message, or add the number to their contacts. Desktop browsers may launch a dialog box to switch apps (for example, to FaceTime on Safari), or they may ignore the link.

If you don't want any interruption on desktop browsers, you could use a CSS rule that hides the link for nonmobile devices (unfortunately, that is beyond the scope of this discussion).

There are a few best practices for using telephone links:

- It is recommended that you include the full international dialing number, including the country code, for the **tel:** value because there is no way of knowing where the user will be accessing your site.
- Also include the telephone number in the content of the link so that if the link doesn't work, the telephone number is still available.
- Android and iOS have a feature that detects phone numbers and automatically turns them into links. Unfortunately, some 10-digit numbers that are not telephone numbers might get turned into links, too. If your document has strings of numbers that might get confused as phone numbers, you can turn autodetection off by including the following **meta** element in the **head** of your document:

  ```
  <meta name="format-detection" content="telephone=no">
  ```

  This will also prevent the browser from overriding any styles you've applied to telephone links.

TEST YOURSELF

The most important lesson in this chapter is how to write URLs for links and images. Here's another chance to brush up on your pathname skills.

Take a look at the directory hierarchy shown in FIGURE 6-12. This diagram should provide you with enough information to write out the markup for links and images described in each of the following questions. I filled in the first one for you as an example, and you can find answers for the rest in **Appendix A**. If you need to work in a more hands-on manner to figure them out, the directory structure is available in the *test* directory in the *materials* folder for this chapter. The documents are just dummy files and contain no content.

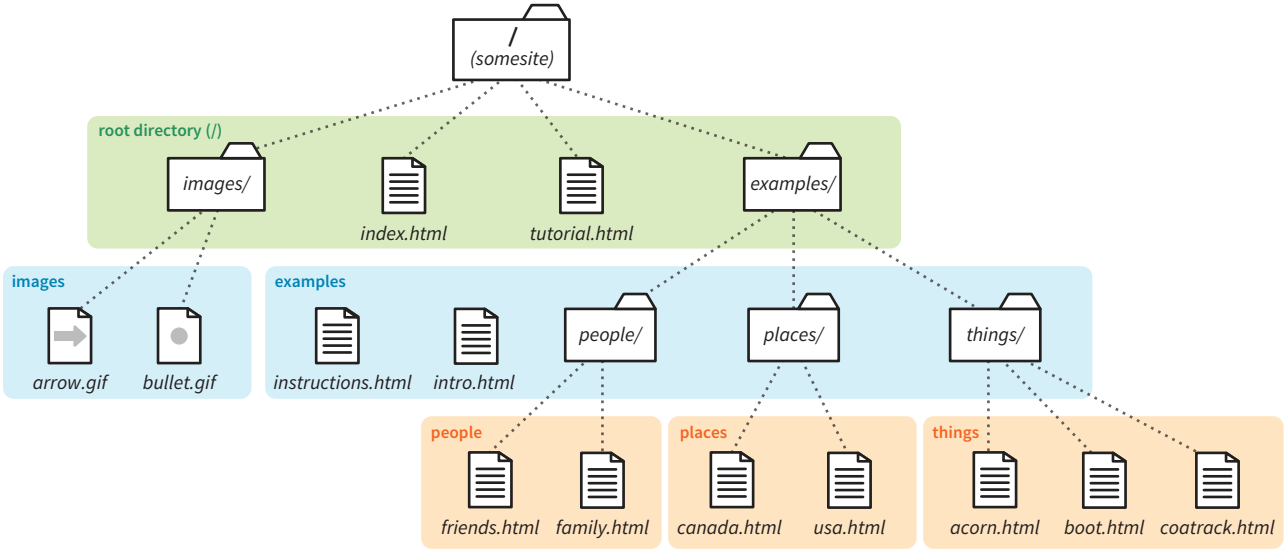


FIGURE 6-12. The directory structure for the "Test Yourself" questions

1. In *index.html* (the site's home page), write the markup for a link to the *tutorial.html* page.

   ```
   <a href="tutorial.html">...</a>
   ```

2. In *index.html*, write the anchor element for a link to *instructions.html*.

3. Create a link to *family.html* from the page *tutorial.html*.

4. Create a link to *boot.html* from the *family.html* page, but this time, start with the root directory.

5. Create a link back to the home page (*index.html*) from *instructions.html*.

6. Create a link to the website for this book (***learningwebdesign.com***) in the file *intro.html*.

7. Create a link to *instructions.html* from the page *usa.html*.

8. Create a link back to the home page (*index.html*) from *acorn.html*.

We haven't covered the image (**`img`**) element in detail yet, but you should be able to fill in the relative URLs after the **`src`** attribute to specify the locations of the image files for the rest of these examples.

9. To place the graphic *arrow.gif* on the *index.html* page, use this URL:

```
<img src="                                  " alt="arrow pointing right">
```

10. To place the graphic *arrow.gif* on the *intro.html* page, use this URL:

```
<img src="                                  " alt="arrow pointing right">
```

11. To place the graphic *bullet.gif* on the *friends.html* page, use this URL:

```
<img src="                                  " alt="small red dot">
```

ELEMENT REVIEW: LINKS

There's only one element relevant to creating hypertext links.

Element and attributes	Description
`a`	Anchor (hypertext link) element
`href="`*`URL`*`"`	Location of the target file
`target="`*`text string`*`"`	Targets a browser window by name
`rel="`*`keyword`*`"`	Specifies the relationship between the current document and the linked document (available *keyword* values are **`alternate`**, **`author`**, **`bookmark`**, **`external`**, **`help`**, **`license`**, **`next`**, **`nofollow`**, **`noreferrer`**, **`noopener`**, **`prev`**, **`search`**, and **`tag`**)

CHAPTER 7

ADDING IMAGES

IN THIS CHAPTER

Adding images with the img element

Image accessibility

Adding SVG images

Providing multiple formats with the picture element

The web's explosion into mass popularity was due in part to the fact that there were images on the pages. Before images, the internet was a text-only tundra.

Images appear on web pages in two ways: embedded in the content or as background images. If the image is part of the editorial content, such as a product shot, gallery image, ad, logo, illustration, and so on, then it should be placed in the flow of the HTML document. If the image is purely decorative, such as a stunning photograph in the background of the header or a patterned border around an element, then it should be added as a background image with Cascading Style Sheets. Not only does it make sense to put images that affect presentation in a style sheet, but it makes the document cleaner, more accessible, and much easier to update later.

I'll talk about CSS background images at length in **Chapter 13, Colors and Backgrounds**. This chapter focuses on embedding image content into the flow of the document, and it is divided into three parts. First, we'll look at the tried-and-true `img` element for adding basic images to a page the way we've been doing it since 1992. It has worked just fine for over 30 years, and as a beginner, you'll find it meets most of your needs as well.

The second part of this chapter introduces some of the methods available for embedding **SVG** (**Scalable Vector Graphics**) images in HTML documents. SVGs are a special case and demand special attention.

Finally, we'll look at the `picture` element that allows us to take advantage of newer image file formats while providing fallbacks for older browsers. I'll also briefly address providing appropriate images for various screen sizes (a technique commonly called **responsive images**).

FIRST, A WORD ON IMAGE FORMATS

We'll get to the `img` element and other markup examples in a moment, but first, it's important to know that you can't put just any image on a web page; it needs to be in one of the web-supported formats.

In general, images that are made up of a grid of colored pixels (called **bitmap** or **raster images**, as shown in FIGURE 7-1, top) must be saved in the PNG, JPEG, WebP, or GIF file formats in order to be placed inline in the content. Newer, more optimized AVIF and JPEG XL bitmap image formats are slowly gaining in popularity, particularly now that we have the `picture` element to make them available to browsers that support them.

For **vector** images (FIGURE 7-1, bottom), such as icons and illustrations you create with tools like Figma or Adobe Illustrator, we have the SVG format. These images are made up of mathematically defined paths, enabling them to be scaled up and down with no loss in quality. There is so much to say about SVGs and their features that they have their own chapter (**Chapter 28, SVG**), but we'll look at how to add them to HTML documents later in this chapter.

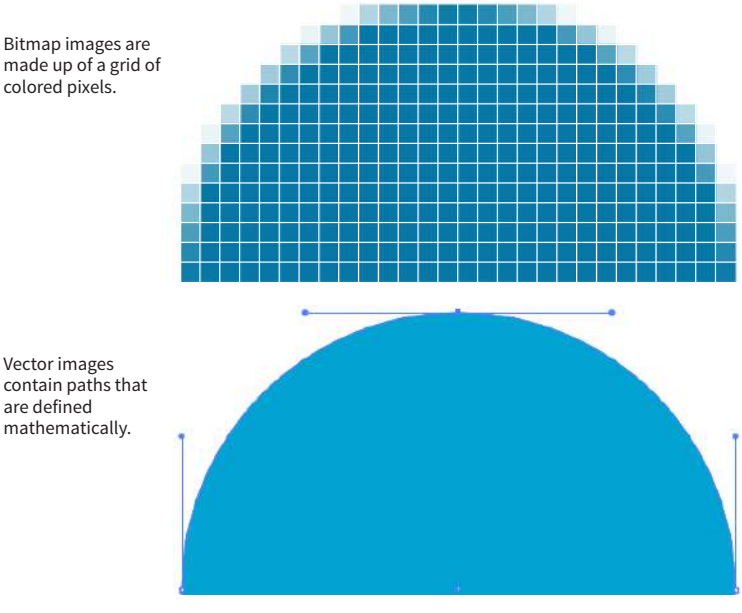


FIGURE 7-1. A comparison of circles saved in bitmapped and vector formats

You should name your image files with the proper suffixes—*.png*, *.jpg* (or *.jpeg*), *.gif*, *.webp*, *.avif*, and *.jxl*, respectively. In addition, your server must be configured to recognize and serve these various image types properly. All web server software today is configured to handle the PNG, JPEG, and GIF formats out of the box, and most now are configured to handle WebP and SVG as well. However, if you're using one of the newer formats, you may need to deliberately add that media type to the server's official list.

A little background information may be useful here. Image files, and indeed any media files that may reside on a server, have an official, standardized media type (also called a **MIME type**; see the sidebar **"File (MIME) Types"** later in this chapter) and suffixes. For example, SVG has the MIME type *image/svg+xml* and the suffixes *.svg* and *.svgz*. Server packages have different ways of handling MIME information. The popular Apache server software uses a file in the root directory called *.htaccess* that contains a list of all the file types and their acceptable suffixes. You might need to add (or ask someone to add) the MIME types of newer image formats so they can be served correctly.

The server looks up the suffix (*.webp*, for example) of a requested image file and matches it with the Content-Type (*image/webp*) that it includes in its HTTP response to the browser. That tells the browser what kind of data is coming and how to parse it.

Images in Other Formats

If you have a source image that is in another popular format, such as TIFF, BMP, or EPS, you'll need to convert it to a web format before you can add it to the page. If, for some reason, you must keep your graphic file in its original format (for example, a file for a CAD program), you can make it available as a downloadable image by making a link directly to the image file, like this:

```
<a href="architecture.eps">Get the drawing</a>
```

Browsers use helper applications to display media they can't handle alone. The browser matches the suffix of the file in the link to the appropriate helper application. An external downloaded image may open in a separate application window, or within the browser window if the helper application is a browser plug-in. The browser may also ask the user to save the file or open an application manually. It is also possible that it won't be able to be opened at all.

Without further ado, let's take a look at the **img** element and its required and recommended attributes.

THE IMG ELEMENT

`<img>`
Adds an inline image

The **img** element tells the browser, "Place an image here." You've already gotten a glimpse of it used to place a banner graphic in the examples in **Chapter 4, Creating a Simple Page**. You can also place an image element right in the flow of the text at the point where you want the image to appear, as in the following example. Images stay in the flow of text, aligned with the baseline of the text, and do not cause any line breaks (the HTML spec calls this a **phrasing element**), as shown in FIGURE 7-2:

```
<p>This summer, try making pizza <img src="pizza.png" alt="photo of a
pizza"> on your grill.</p>
```

FIGURE 7-2. By default, images are aligned with the baseline of the surrounding text and do not cause a line break.

When the browser encounters an `img` element, it makes a request to the server and retrieves the image file before displaying it on the page. On a fast network with a fast computer or device, even though a separate request is made for each image file, the page usually appears to assemble instantaneously. On mobile devices or computers with slow network connections (like the expensive WiFi at luxury hotels), we may be well aware of the wait as images are fetched one at a time.

The `src` and `alt` attributes shown in the example are required in order for the image to appear and the document to be valid. The `src` (source) attribute provides the location of the image file (its URL). The `alt` attribute provides alternative text that displays if the image is not available.

I'll talk about `src` and `alt` more in the following sections, but first, let's recap the key points to note about the `img` element:

- It is a void (or empty) element, which means it doesn't have any content. You just place it in the flow of text where the image should go.
- The `img` element is what's known as a **replaced element** because it is replaced by an external file when the page is displayed. This makes it different from text elements that have their content right there in the source (and thus are **non-replaced**).
- It is a phrasing (inline) element, so it behaves like any other phrasing element in the text flow. FIGURE 7-3 demonstrates the inline nature of image elements. When the browser window is resized, a line of images reflows to fill the new width.
- By default, the bottom edge of an image aligns with the baseline of text, as shown in FIGURE 7-2. Using CSS, you can float the image to the right or left margin and allow text to flow around it, crop it to a shape, control the space and borders around the image, and change its vertical alignment. I'll talk about those styles in **Part III, CSS for Presentation**.

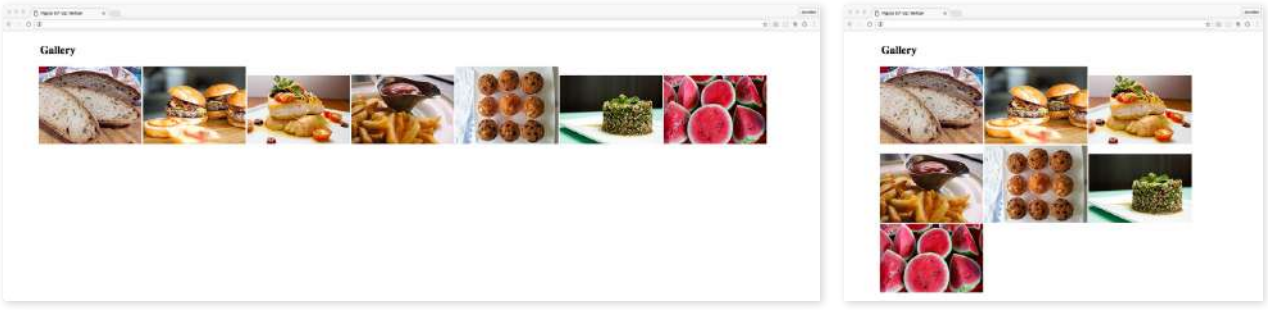


FIGURE 7-3. Inline images are part of the normal document flow. They reflow when the browser window is resized.

Providing the Location with src

`src="URL"`
Source (location) of the image

The value of the **src** attribute is the URL of the image file. In most cases, the images you use on your pages will reside on your own server, so you will use relative URLs to point to them.

If you just read **Chapter 6, Adding Links**, you should be pretty comfortable with writing relative URLs. In short, if the image is in the same directory as the HTML document, you can simply refer to the image by name in the **src** attribute:

```
<img src="icon.gif" alt="">
```

Developers usually organize the images for a site into a directory called *images* or *img* (in fact, organizing them that way can indirectly help search engines crawl and index your files). There may even be separate image directories for each section of the site. If an image is not in the same directory as the document, you need to provide the pathname to the image file:

```
<img src="/images/arrow.gif" alt="">
```

Of course, you could place images from other websites on your page by using a full URL, like this, but it is not recommended (see **Warning**):

```
<img src="http://www.example.com/images/smile.gif" alt="">
```

WARNING

Before you use any image on your web page, be sure that you own the image, that you have explicit written permission from the copyright holder, or that it is in the public domain. Linking to an image on another server (called hotlinking) is considered seriously uncool, so don't do it unless there is a specific use case for which you have permission. Even then, be aware that you cannot control the image and risk it being moved or renamed, which will break your link. You may also find that the server blocks requests that don't come from its own domain (also known as the page's "origin").

■ PERFORMANCE TIP

Take Advantage of Caching

When a browser downloads an image, it stores the file in the disk **cache** (a space for temporarily storing files on the hard disk). That way, if it needs to redisplay the page, it can just pull up a local copy of the image without making a new server request.

If you use the same image repeatedly, be sure that the **src** attribute for each **img** element points to the same URL on the server. That way, the image downloads once, then gets pulled from the cache for subsequent uses. This means less traffic for the server and faster display for the user.

Providing Alternative Text with alt

`alt="text"`
Alternative text

Every **img** element must also contain an **alt** attribute that provides a text alternative to the image for those who are not able to see it. **Alternative text** (also called **alt text**) should serve as a substitute for the image content, conveying the same information and function. Alternative text is used by screen readers, search engines, and graphical browsers when the image doesn't load. There are no hard and fast rules on length; it can be shorter depending on the context but is generally in the 100 to 250 character range, with the best practice of keeping it in the lower end of the range (see **Note**).

NOTE

For longer descriptions, put the image in a **figure** *and use the* **figcaption** *element to provide a detailed description. This may be useful or necessary for infographics or charts that convey a lot of information visually.*

There is an art to writing useful alt text. Consider how it will sound when read aloud and how well it captures the essence and purpose of the image it stands in for.

Provide a full description of the image if it is a photograph, illustration, chart, or infographic. It is a best practice to write longer descriptions in sentence form, including using punctuation so a screen reader knows where to pause and transition to the surrounding page text. For a pie chart, you might write, "The pie chart shows energy sources in percentages: 40% coal, 30% natural gas, 30% renewable." Also consider your audience; for example, the alt text for a photograph of a dog might differ depending on whether the image appears on dog lover's site or a site selling camera equipment.

If the image provides functionality, such as an image used as a link, then the alt text should convey its purpose even if it doesn't describe the image exactly. In the following example, a PDF icon indicates that the link downloads a file in PDF format. In this case, the image is conveying valuable content that will be missing if the image cannot be seen. Providing the alt text "PDF file" replicates the purpose of the image better than "an icon depicting a folded piece of paper":

```
<a href="application.pdf">High school application</a> <img src="images/
pdflogo.png alt="PDF file">
```

A screen reader might indicate the image by reading its **alt** value this way:

"High school application. Image: PDF file."

Sometimes images function as links, in which case providing alternative text is critical because screen readers need something to read for the link. In the next example, an image of a book cover is used as a link to the book's website. Its alt text does not describe the cover itself, but rather performs the same function as the cover image on the page (indicating a link to the site):

```
<a href="https://learningwebdesign.com"><img src="/images/LWD_cover.
png" alt="Learning Web Design site"></a>
```

If an image is purely decorative and does not add anything meaningful to the text content of the page, you still need to include the **alt** attribute, but it is recommended that you leave the value empty (**null**). In the following example, a decorative floral accent is not contributing to the content of the page, so

■ ACCESSIBILITY TIP

If the medium of the image—for example, painting, photograph, or illustration—is relevant to the content, then include it in the description, as in "A photograph of a puppy playing with a red ball" or "A rough pencil sketch of a man's face." Avoid the more generic "Image of…" because most screen readers announce that it is an image or a graphic before reading the alternative text.

its **alt** value is null. (You may also consider whether it is more appropriately handled as a background image in CSS.) Note that there is no character space between the quotation marks:

```
<img src="/images/floralembellishment.png" alt="">
```

For each inline image on your page, consider what the alternative text would sound like when read aloud and whether that enhances the experience or might be obtrusive to a user with assistive technology.

Alternative text may benefit users with visual browsers as well. If the image fails to load, the browser may display this text to give the user an idea of what is missing. The handling of alternative text is inconsistent among modern browsers, however, as shown in FIGURE 7-4.

Alternative text is important to accessibility and warrants more exploration. In addition to taking a look at the HTML spec's alt text recommendations (*html.spec.whatwg.org/multipage/images.html#alt*), I recommend the following articles, both of which have plenty of links to additional resources:

- "How to Write Alt Text and Image Descriptions for the Visually Impaired," published by the Perkins School for the Blind (*www.perkins.org/resource/how-write-alt-text-and-image-descriptions-visually-impaired*)
- "Write Alt Text Like You're Talking to a Friend," by Scott Vandehey (*cloudfour.com/thinks/write-alt-text-like-youre-talking-to-a-friend/*)

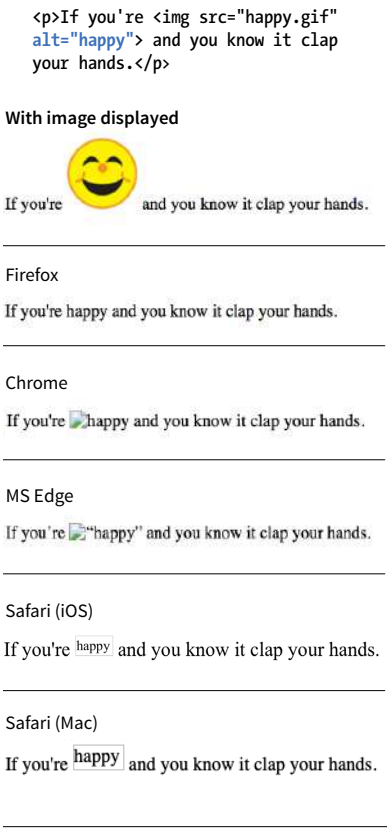

If you're <img src="happy.gif"
alt="happy"> and you know it clap
your hands.
With image displayed
If you're
and you know it clap your hands.
Firefox
If you're happy and you know it clap your hands.
Chrome
If you're happy and you know it clap your hands.
MS Edge
If you're "happy" and you know it clap your hands.
Safari (iOS)
If you're happy and you know it clap your hands.
Safari (Mac)
If you're happy and you know it clap your hands.

FIGURE 7-4. Browsers display alternative text in place of the image if the image is not available.

■ FURTHER READING

Image Accessibility

Some types of images, such as data charts and diagrams, require long descriptions that aren't practical as **alt** values. These cases require alternate accessibility strategies, which you can find out more about in these resources:

- The WebAIM article "Accessible Images" (*webaim.org/techniques/images/*).
- The WebAIM article "Alternative Text" (*webaim.org/techniques/alttext/*).
- The W3C's Web Content Accessibility Guidelines (*w3.org/TR/WCAG20-TECHS/*). These include techniques for improving accessibility across all web content. Warning: this one is pretty dense.

Providing the Dimensions with width and height

The **width** and **height** attributes indicate the dimensions of the image in number of pixels. Without these dimensions specified, the browser doesn't know how much space to set aside for images, and the result is that the content of the page will shift around once the images finally do arrive—a frustrating user experience for sure!

`width="number"`
Image width in pixels

`height="number"`
Image height in pixels

Be sure that the pixel dimensions you specify are the actual dimensions of the image. If the pixel values differ from the actual dimensions of your image,

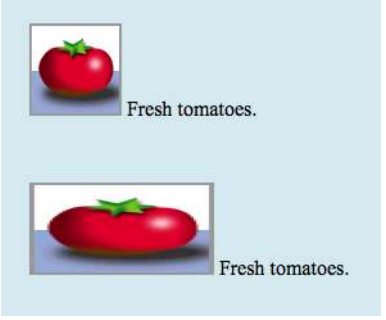


FIGURE 7-5. Browsers resize images to match the provided **width** and **height** values, but you should not resize images this way.

> **■ PERFORMANCE TIP**
>
> It's important to note that the **width** and **height** attributes only change the size the image is displayed at in the browser—they don't change the size of the image file itself. You should try to make sure the actual image file is sized reasonably close to the dimensions it will be displayed at, to make sure your visitors aren't spending potentially limited data downloading much larger images than they actually need.

the browser resizes the image to match the specified values (FIGURE 7-5). If you are using **width** and **height** attributes and your image looks distorted or even slightly blurry, check to make sure that the values are in sync. If only one dimension is set, the image scales proportionally.

The majority of the time, the display size of images is controlled using the CSS **width** and **height** properties. However, it is still a best practice to include the **width** and **height** attributes, because modern browsers use the values to establish the aspect ratio of the image and keep it proportional even when the size is overridden by CSS rules.

Lazily Loading Images with the loading Attribute

`loading="lazy | eager"`
Specifies the loading priority for the image

Lazy loading is a strategy for ensuring that resources such as images are loaded only when they're needed. An image toward the top of the page, for example, is important to load right away, but images that appear later on in the page, after the user has scrolled down, are not.

When a browser downloads the HTML document for a web page, the first thing it does is look through the whole document and establish its outline structure (its Document Object Model, or DOM). Then, almost immediately, a **preloader** goes out to get all the images, CSS, and JavaScript that it sees referenced in the HTML. This makes perfect sense for images that are displayed higher in the viewport, but for images that won't be used until and unless your visitors scroll down the page, it isn't necessary to download them so aggressively—in fact, it would much better for performance if they were downloaded later, just before they were going to appear.

The **loading** attribute lets us tell the browser when we think an image may be able to be loaded later on, freeing up the browser to get the rest of the page displayed as quickly as possible.

There are two supported values for the **loading** attribute:

`lazy`

: This value tells the browser the image does not need to be downloaded right away. The browser will instead wait to load the image until it's about to come into view.

`eager`

: This is the default loading behavior, and it's what the browser does if there is no **`loading`** attribute specified. **`eager`** tells the browser to download this image right away, regardless of where it sits on the page.

Using the **`loading`** attribute is a great way to improve performance and help your visitors use less data, but you do want to be careful to only lazy load images that do not appear in the initial viewport. For example, let's say we have a large image near the top of our page:

```
<img src="/images/floralembellishment.png" loading="lazy">
```

Because we told the browser to load this image using a lazy loading strategy, we'll actually end up making it take longer to display the image. That's what we want for images that appear lower down on in the page, but it's not what we want for images that are displayed higher up. In this case, we would be much better off either applying the eager strategy or omitting the loading attribute entirely:

```
<img src="/images/floralembellishment.png" loading="eager">
```

Now that you know the basics of the **`img`** element, you should be ready to add a few photos to the Black Goose Bistro Gallery site in EXERCISE 7-1.

ADDING SVG IMAGES

No lesson on adding images to web pages would be complete without an introduction to adding Scalable Vector Graphics. After all, the popularity of SVG images has been gaining momentum, thanks to nearly ubiquitous browser support and the need for images that can resize without loss of quality. For illustration-style images, they are a responsive dream come true. I'm saving our deep dive into all things SVG for **Chapter 28**, but for now I'll give you a quick peek at what they're made of so that the embedding markup makes sense.

As I mentioned at the beginning of this chapter, SVGs are an appropriate format for storing vector images (FIGURE 7-1). Instead of a grid of pixels, vectors are made up of shapes and paths that are defined mathematically. Even more interestingly, in SVGs those shapes and paths are specified by instructions written out in a text file. Let that sink in: they are *images* that are written out in *text*! All of the shapes and paths, as well as their properties, are written out in the standardized SVG markup language (see **Note**). Just as HTML has elements for paragraphs (**`p`**) and tables (**`table`**), SVG has elements that define shapes like rectangles (**`rect`**), circles (**`circle`**), and paths (**`path`**).

NOTE

SVG is an example, or application, of XML (Extensible Markup Language), which provides the rules and standards for how markup languages should be written and work together. As a result, SVG plays well alongside HTML content, which is also based on XML.

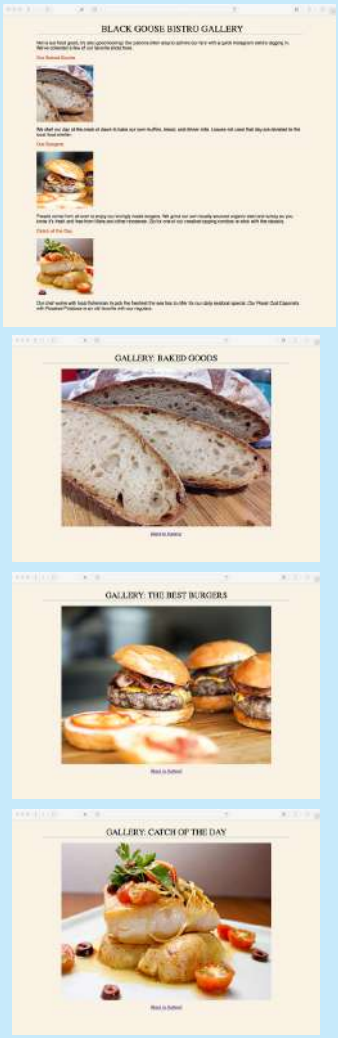


FIGURE 7-6. The finished pages for the Black Goose Bistro Gallery

EXERCISE 7-1. Adding and linking images

In this exercise, you'll add images to pages and use them as links. All of the full-size photos and thumbnails (small versions of the images) you need have been created for you, and I've given you a head start on the HTML files with basic styles as well. The starter files and the resulting code are available at *learningwebdesign.com/6e/materials.* Put a copy of the *gallery* folder on your hard drive, making sure to keep it organized as you find it.

This little site is made up of a main page (*index.html*) and three separate HTML documents containing the larger images (FIGURE 7-6). First we'll add the thumbnails, and then we'll add the full-size versions to their respective pages. Finally, we'll make the thumbnails link to those pages. Let's get started.

Open the file *index.html*, and add the small thumbnail images to this page to accompany the text. I've done the first one for you:

```
<img src="thumbnails/bread-200.jpg" alt="Photograph of a close-up of
sliced rustic bread" width="200" height="200">
<p>We start our day at the...
```

I've put the image just before the paragraph. Because all of the thumbnail images are located in the *thumbnails* directory, I provided the pathname in the URL. I added a description of the image with the **alt** attribute and supplied the **width** and **height** attributes as well, to tell the browser how much space to leave in the layout. Now it's your turn:

1. Add the thumbnail images *burgers-200.jpg* and *fish-200.jpg* before the paragraphs in their respective sections, following my example. Be sure to include the pathnames and thoughtful alternative text descriptions.

   When you are done, save the file and open it in your browser to be sure that the images are visible and appear at the right size.

2. Next, add the images to the individual HTML documents. I've done *bread.html* for you:

   ```
   <h1>Gallery: Baked Goods</h1>
   <p><img src="photos/bread-800.jpg" alt="Photograph of a close-up of
   sliced rustic bread " width="800" height="600"></p>
   ```

   The full-size images are in a directory called *photos*, so that needs to be reflected in the pathnames, and I included the **width** and **height** attributes here to establish the aspect ratio of the image. Note that I've included a style rule in all of the photo pages that makes the image resize to fit the width of the browser window (try making your browser window wider and narrower). It's responsive!

   Add images to *burgers.html* and *fish.html*, following the example. Hint: all of the images are 800 pixels wide and 600 pixels high.

   Save each file, and check your work by opening them in the browser window.

3. Back in *index.html*, link the thumbnails to their respective files. I've done the first one:

   ```
   <p><a href="bread.html"><img src="thumbnails/bread-200.jpg" alt="close-
   up of sliced rustic bread" width="200" height="200"></a><br>We start our
   day at the crack of dawn...
   ```

   Notice that the URL is relative to the current document (*index.html*), not to the location of the image (the *thumbnails* directory).

Make the remaining thumbnail images link to each of the documents. If all the images are visible and you are able to link to each page and back to the home page again, then congratulations, you're done!

Like More Practice?

If you'd like more practice, you'll find three additional images (*chicken-800.jpg*, *fries-800.jpg*, and *tabouleh-800.jpg*) with their thumbnail versions (*chicken-200.jpg*, *fries-200.jpg*, and *tabouleh-200.jpg*) in the appropriate directories. You'll need to add your own descriptions to the home page and create the HTML documents for the full-size images from scratch.

A simple example will give you the general idea. Here is the SVG code that describes a rectangle (**rect**) with rounded corners (**rx** and **ry**, for x-radius and y-radius) and the word "hello" set as **text** with attributes for the font and color (FIGURE 7-7). Browsers that support SVG read the instructions and draw the image exactly as it was designed:

```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 300 180">
  <rect width="300" height="180" fill="purple" rx="20" ry="20"/>
  <text x="40" y="114" fill="yellow" font-family="'Verdana-Bold'"
font-size="72">hello!</text>
</svg>
```

FIGURE 7-7. A simple SVG made up of a rectangle and text

SVGs offer some significant advantages over their bitmapped counterparts for certain image types:

- Because they save only instructions for what to draw, they generally require less data than an image saved in a bitmapped format. That means faster downloads and better performance.
- Because they are vectors, they can resize as needed in a responsive layout without loss of quality. An SVG is always nice and crisp, no fuzzy edges.
- Because they are text, they integrate well with HTML/XML and can be compressed with tools like Gzip and Brotli, just like HTML files.
- They can be animated.
- You can change how they look with Cascading Style Sheets.
- You can add interactivity and animation with JavaScript.

Again, all of the ins and outs of creating SVGs, as well as their many features, are discussed in detail in **Chapter 28**. I'll be focusing on the HTML required to place them in the content flow. You have a few options—SVGs can be:

- Embedded with the **img** element
- Written out in code as an inline **svg** element
- Embedded with the **object** element
- Used as a background image with CSS

Embedded with the img Element

An SVG text file saved with the *.svg* suffix (sometimes referred to as a **standalone SVG**) can be treated like any other image, including placing it in the document by using the **img** element. You're an expert on the **img** element by now, so the following example should be clear:

```
<img src="/images/circle.svg" alt="">
```

NOTE

*For very old browers that do not support the SVG format, you could use the picture element to provide a bitmap fallback. See "***The picture Element for Alternative Image Format***" section for details.*

Pros and cons

The advantage to embedding an SVG with **img** is that it is universally supported in browsers that support SVG (see **Note**).

This approach works fine when you are using a standalone SVG as a simple substitute for a GIF or a PNG, but there are a few disadvantages to embedding SVGs with **img**:

- You cannot apply styles to the items within the SVG with an external style sheet, such as a *.css* file applied to the whole page. The *.svg* file may include its own internal style sheet using the **style** element, however, for styling the elements within it.
- You cannot manipulate the elements within the SVG with JavaScript, so you lose the option for interactivity. Scripts in your web document can't see the content of the SVG, and scripts in the SVG file do not run at all when the SVG is placed with an **img** element. Other interactive effects, like links and **:hover** styles, are also never triggered inside an SVG embedded with **img**.
- You can't use *any* external files, such as embedded images or web fonts, within the SVG.

In other words, standalone SVGs behave as though they are in their own little self-contained bubbles. For static illustrations, that's just fine.

SVG Server Configuration

If you are using SVGs and they are not showing up correctly when your site is uploaded, you may need to configure the server to recognize the SVG image type, as discussed at the beginning of this chapter. Here's how to do it on the Apache server, but similar configurations can be done in other server languages:

```
AddType image/svg+xml .svg
```

Inline in the HTML Source

<svg>...</svg>
An inline SVG image

Another option for putting an SVG on a web page is to copy the content of the SVG file and paste it directly into the HTML document. This is called using the SVG **inline**. Here is an example that looks a lot like the inline **img** example that we saw way back in FIGURE 7-2, only this time the pizza is a vector image drawn with circles and inserted with the **svg** element (FIGURE 7-8). Each **circle** element has attributes that describe the fill color, the position of its center point (**cx** and **cy**), and the length of its radius (**r**):

```
<p>This summer, try making pizza
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 72 72" width="100"
height="100">
  <circle fill="#D4AB00" cx="36" cy="36" r="36"/>
  <circle opacity=".7" fill="#FFF" stroke="#8A291C" cx="36.1" cy="35.9"
r="31.2"/>
```

```
    <circle fill="#A52C1B" cx="38.8" cy="13.5" r="4.8"/>
    <circle fill="#A52C1B" cx="22.4" cy="20.9" r="4.8"/>
    <circle fill="#A52C1B" cx="32" cy="37.2" r="4.8"/>
    <circle fill="#A52C1B" cx="16.6" cy="39.9" r="4.8"/>
    <circle fill="#A52C1B" cx="26.2" cy="53.3" r="4.8"/>
    <circle fill="#A52C1B" cx="42.5" cy="27.3" r="4.8"/>
    <circle fill="#A52C1B" cx="44.3" cy="55.2" r="4.8"/>
    <circle fill="#A52C1B" cx="54.7" cy="42.9" r="4.8"/>
    <circle fill="#A52C1B" cx="56" cy="28.3" r="4.8"/>
  </svg>
on your grill.</p>
```

FIGURE 7-8. This pizza image is an SVG made up of 11 `circle` elements. Instead of an `img` element, the SVG source code is placed right in the HTML document with an `svg` element.

This code was generated by Adobe Illustrator, where I created the illustration and saved it in SVG format. I also optimized it to strip out a lot of cruft that Illustrator adds in there. I'll discuss SVG optimization in **Chapter 28**.

Pros and cons

Inline SVGs allow developers to take full advantage of SVG features. When the SVG markup is alongside the HTML markup, all of its elements are part of the main DOM tree. That means you can access and manipulate SVG objects with JavaScript, making them respond to user interaction or input. There are similar benefits with regard to style sheets: the elements in the SVG can inherit styles from HTML elements, which makes it easy to apply the same styles to elements on the page and within the SVG graphic. The good news is that all modern browsers support SVG images placed inline with the `svg` element.

On the downside, the code for SVG illustrations can get extremely long and unwieldy, resulting in bloated HTML documents that are difficult to read. Even that little pepperoni pizza requires a serious block of code. It also makes the images for a site more difficult to maintain, because they are tucked away in the HTML documents. Another disadvantage is that inline SVGs are not cached by the browser separately from the HTML file, so avoid this method for large images that are reused across many HTML pages.

Embedded with the object Element

HTML has an all-purpose media embedding element called **object**. I'll talk about it more in **Chapter 10, Embedded Media**, but for now, know that **object** is another option for embedding an SVG in a web page. It's a good compromise between **img** and inline SVG, allowing a fully functional SVG that is still encapsulated in a separate, cacheable file.

The opening **object** tag specifies the media type (an **svg+xml** image) and points to the file to be used with the **data** attribute. The **object** element comes with its own fallback mechanism: any content within the **object** gets rendered if the media specified with **data** can't be displayed. In this case, a PNG version of the image will be placed with an **img** if the *.svg* format is not supported or the image fails to load:

```
<object type="image/svg+xml" data="pizza.svg">
  <img src="pizza.png" alt="pizza">
</object>
```

There is one catch, however: all modern browsers download the fallback image, even though they support SVG and don't need it. Useless downloads are not ideal. The workaround is to make the fallback image a CSS background image in an empty **div** container. Unfortunately, this isn't as flexible for scaling and sizing, but it does solve the extra download issue:

```
<object type="image/svg+xml" data="pizza.svg">
  <div style="background-image: url(pizza.png); width 100px; height:
100px;" role="img" aria-label="pizza">
</object>
```

Pros and cons

The main advantage to embedding SVGs with the **object** element is that they can load external files. They can also use scripts to access the parent HTML document (with some security restrictions). However, because they are separate files and not part of the DOM for the page, you can't use a style sheet in the HTML document to style elements within the SVGs. Embedded SVGs may also have some buggy behaviors in browsers and can cause unnecessary downloads, so be sure to test thoroughly. Generally speaking, you'll want to opt for one of the other approaches to loading SVG images in most cases.

Other Embedding Options

Older techniques for adding SVGs involve using two other HTML elements for embedding media: **embed** and **iframe** (we'll talk about them in **Chapter 10**). You may still see these in use with SVGs out there, and they work fine for browsers that support SVG, but most developers consider them to be outdated methods. Stick with **img**, inline **svg**, **object**, and CSS **background-image**.

Used as a Background Image with CSS

This is an HTML chapter, but I'd be remiss if I didn't at least mention that SVGs can be used as background images with CSS. This style rule example puts a decorative image in the background of a **header**:

```
header {
  background-image: url(/images/decorative.svg);
}
```

Are you ready to give SVGs a spin? Try out some of the embedding techniques we discussed in EXERCISE 7-2.

EXERCISE 7-2. Adding an SVG to a page

In this exercise, we'll add some SVG images to the Black Goose Bistro page that we worked on in **Chapter 4**. The materials for this exercise are available online at *learningwebdesign.com/6e/materials*. You will find everything in a directory called *svg*. This exercise has two parts: first, we'll replace the logo with an SVG version, and second, we'll add a row of social media icons at the bottom of the page (FIGURE 7-9).

Part I: Replacing the logo

1. Open *blackgoosebistro.html* in a text editor. It should look like we left it in **Chapter 4**.
2. Just for fun, let's see what happens when you make the current PNG logo really large. Add `width="500" height="500"` to the `img` tag. Save the file and open it in the browser to see how blurry bitmap images get when you size them larger. Yuck.
3. Let's replace it with an SVG version of the same logo by using the inline SVG method. In the *svg* folder, you will find a file called *blackgoose-logo.svg*. Open it in your text editor and copy all of the text (from `<svg>` to `</svg>`).
4. Go back to the *blackgoosebistro.html* file and delete the entire `img` element (be careful not to delete the surrounding markup). Paste the SVG text in its place. If you look closely, you will see that the SVG contains two circles, a gradient definition, and two paths (one for the starburst shape and one for the goose).
5. Set the size the SVG should appear. In the opening `<svg>` tag, add `width` and `height` attributes set to 200 pixels (because pixels are the default unit for SVG, you can omit the `px`):

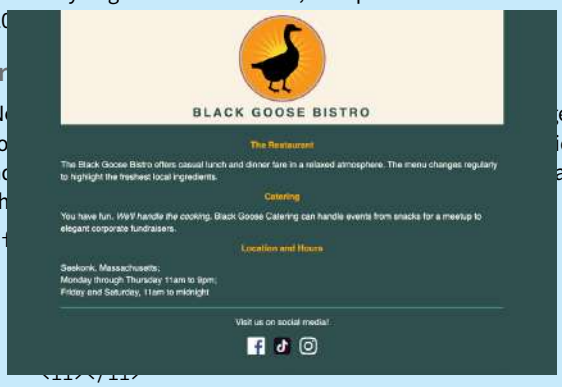
   ```
   <h1><svg width="200" height="200" ...
   ```

6. Save the file and open the page in the browser. You should see the SVG logo in place, looking a lot like the old one.
7. Try seeing what happens when you make the SVG logo really big. Change the width and height to 500 pixels, save the file, and reload the page in the browser. It should be big and *sharp*! No blurry edges like the PNG. OK, now put the size back to 200 × 20

Par

8. N… e fo…ion, a… add th…

   <f…


BLACK GOOSE BISTRO
The Restaurant
The Black Goose Bistro offers casual lunch and dinner fare in a relaxed atmosphere. The menu changes regularly to highlight the freshest local ingredients.
Catering
You have fun. We'll handle the cooking. Black Goose Catering can handle events from snacks for a meetup to elegant corporate fundraisers.
Location and Hours
Seekonk, Massachusetts;
Monday through Thursday 11am to 9pm;
Friday and Saturday, 11am to midnight
Visit us on social media!

```
    </ul>
  </footer>
```

FIGURE 7-9. The Black Goose Bistro page with SVG images

9. Use the `img` element to place three SVG icons: *facebook.svg*, *tiktok.svg*, and *instagram.svg*. They are located in the *icons* directory. Here's a head start on the first one:

   ```
   <li><img src="icons/facebook.svg"
   alt="Facebook"></li>
   ```

   Ideally, each icon would serve as a link, but we don't need to worry about that in this exercise.

10. Save the file and open it in the browser (FIGURE 7-9). I've included style rules in the starter document (*blackgoosebistro.html*) that resize the icons and make them display in a row. If you started this exercise with your own document from the previous chapter, you will need to add the following rules to the `style` element. This may be unfamiliar, but rest assured you'll understand these rules in just a few chapters:

    ```
    footer {
      border-top: 1px solid turquoise;
      text-align: center;
      margin-top: 2em;
    }
    footer img {
      width: 40px;
      height: 40px;
    }
    ul {
      list-style-type: none;
      margin: 0;
      padding: 0;
    }
    li {
      display: inline;
      margin: 0 15px;
    }
    ```

Go ahead and play around with the style settings, or even the code in the inline goose SVG, if you'd like to get a feel for how they affect the appearance of the images.

THE PICTURE ELEMENT FOR ALTERNATIVE IMAGE FORMATS

`<picture>...</picture>`
Specifies a number of image options

`<source>...</source>`
Specifies alternate image sources

Once upon a time, in the early 1990s, the only image type you could put on a web page was a GIF (I know, I was there). JPEGs came along not long after, then we waited nearly a *decade* for reliable browser support for the more feature-rich PNG format. It takes a notoriously long time for new image formats to become universally supported. In the past, that meant simply avoiding newer formats.

In an effort to reduce image file sizes, more efficient image formats have been developed—such as JPEG XL and AVIF—that can compress images significantly smaller than their JPEG counterparts (see **Note**). And once again, some browsers support them and some don't. The difference is that today we can use the **`picture`** element to serve the newer, smaller image formats to browsers that can handle them, and a standard image format to browsers that can't. We no longer have to wait for universal browser support.

NOTE

The bitmap image formats, including WebP, JPEG XL, and AVIF, are discussed in more detail in **Chapter 26, Web Image Basics**.

The **`picture`** element has no attributes (other than the global attributes); it is just a wrapper for some number of **`source`** elements and an **`img`** element used as a fallback. The **`img`** element is required and must be the last element in the list. If it's left out, no image will display at all, because it is the piece that is actually placing the image on the page.

Each **`source`** element points to the location of the image file with the **`srcset`** attribute and declares the file format with the **`type`** attribute. The value of **`type`** is the MIME type for the image format (see the **"File (MIME) Types"** sidebar).

Browsers download the image from the first **`source`** that the browser supports, passing the URL provided in the **`srcset`** attribute to the **`src`** attribute in the **`img`** element. The **`alt`** attribute is required for the **`img`** element, but **`alt`** is not permitted in the **`source`** element.

In the following example, the **`picture`** element specifies two image alternatives before the fallback JPEG listed in the **`img`** element:

```
<picture>
  <source type="image/avif" srcset="pizza.avif">
  <source type="image/webp" srcset="pizza.webp">
  <img src="pizza.jpg" alt="">
</picture>
```

The first **`source`** points to an image that is in the AVIF format, and the second specifies a WebP. Because the browser uses the image from the first **`source`** that matches its image support, it makes sense to put them in order from smallest to largest file size.

The **`picture`** element is also an important tool in responsive web design for providing different images based on screen size. For example, when the browser window is wide, such as on a desktop monitor, a large image full of

File (MIME) Types

The web uses a standardized system to communicate the type of media files being transferred between the server and browser. It is based on MIME (Multipurpose Internet Mail Extension), which was originally developed for sending attachments via email. Every file format has a standardized type (such as **`image`**, **`application`**, **`audio`**, or **`video`**), a subtype that identifies the specific format, and one or more file extensions. In the **`picture`** code example, the **`type`** attribute specifies the WebP option with its type/subtype (**`image/webp`**) and uses the proper file extension (*.webp*). Other examples of media MIME types are **`image/jpeg`** (extensions *.jpg*, *.jpeg*), **`video/mpeg`** (extensions *.mpg*, *.mpe*, *.mpeg*, *.m1v*, *.mp2*, *.mp3*, and *.mpa*), and **`application/pdf`** (*.pdf*). The complete listing of registered MIME types is published by the Internet Assigned Numbers Authority (IANA) at *iana.org/assignments/media-types*.

detail would be appropriate. For small screens, such as a smartphone, using a cropped version may make more sense than allowing the large version to shrink down to a level where the detail would be lost (and where downloading it would be a waste of data).

This provides a good segue to a brief discussion of responsive images.

A QUICK INTRODUCTION TO RESPONSIVE IMAGES

As I've mentioned before, as web developers, we should always strive to make our pages as light as possible and avoid downloading unnecessary data. There's not much point in providing a gigantic hero image made to fill a desktop browser to a small smartphone, or serving a high-resolution image made to look sharp on high-density displays to a device with standard resolution. Ideally, we should give the browser just as much image data as it needs, and no more.

Shortly after smartphones and tablets exploded on the scene, it quickly became apparent that the `img` element was a crude instrument for putting images on screens of various sizes. One image often doesn't serve all devices well. The browser companies, the W3C, and the web development community responded by working together on HTML markup for selecting the most appropriate image from a set of options tailored to various viewing environments. Taken together, these data-saving techniques are referred to as **responsive images**, and they address four basic scenarios:

- Providing extra-large images that look crisp on **high-resolution screens** with the `srcset` attribute for `img`
- Providing a set of images of various dimensions for use on **different screen sizes** with the `srcset` attribute for `img`
- Providing versions of the image with varying amount of detail based on the device size and orientation with the `picture` element (known as the **art direction** use case)
- Providing **alternative image formats** that store the same image at smaller file sizes with the `picture` element (as we discussed earlier in this chapter)

The way it works is you create several versions of an image and let the browser pick the most appropriate one for the job, based on the rules and information you provide in the markup. Responsive image markup is not as straightforward as the examples we've seen so far in this book. It's based on more advanced web development concepts, and the syntax may be tricky for someone just getting started writing HTML (heck, it's a challenge for seasoned professionals!). For that reason, I've moved that discussion to the article **"Responsive Image Markup,"** available at ***learningwebdesign.com/articles***,

which you can read once you are more familiar with CSS and responsive web design concepts. I recommend saving it for when you're ready. In **Chapter 27, Image Production Techniques**, you'll learn strategies for producing the sets of images themselves.

AND WE'RE FINISHED!

That wraps up our exploration of images. First, we covered how to place images with the `img` element and its required `src` and `alt` attributes. We talked about the importance of good alternative text for accessibility, and how we can help the browser make better decisions about when to load images so our pages are displayed as fast as possible. Next, we looked at a few ways to embed SVG images into a web page. Finally, we looked at how the `picture` and `source` elements can be used as powerful tools for using alternative image formats without waiting for full-browser support. You also got a glimpse into the world of responsive images. Now try answering a few questions to test your knowledge.

TEST YOURSELF

Images are a big part of the web experience. Answer these questions to see how well you've absorbed the key concepts of this chapter. The correct answers can be found in **Appendix A**.

1. Which attributes must be included in every `img` element?

2. Write the markup for adding an image called *furry.jpg* that is in the same directory as the current document.

3. Name two reasons to include alternative text for an `img` element.

4. What might be going wrong if your images don't appear when you view the page in a browser? There are three possible explanations.

5. What is a cache and how does it affect web page performance?

6. Name one advantage and one disadvantage of adding an SVG to a page with the **`img`** element.

7. Name one advantage and one disadvantage of inline SVG.

8. When would it be appropriate to add an SVG to a page as a background image with CSS?

9. What is this bit of code describing, and when might you need to use it?

```
image/svg+xml
```

ELEMENT REVIEW: IMAGES

Here's a review of the elements you learned about in your exploration of image markup.

Element and attributes	Description
`img`	Inserts an inline image
`alt="`*`text`*`"`	Alternative text
`src="`*`url`*`"`	The location of the image file
`srcset="`*`list of urls with descriptors`*`"`	Images to use in different situations
`sizes="`*`list media conditions and layout sizes`*`"`	Image sizes for different layouts
`width="`*`number`*`"`	Horizontal dimension
`height="`*`number`*`"`	Vertical dimension
`picture`	Container that provides multiple sources to its contained **`img`** element

Element and attributes	Description
`source`	Provides alternate sources for the **`img`** element
`src="URL"`	Address of the image resource
`srcset="URL"`	Images to use in different situations
`sizes="source size list"`	Image sizes for different page layouts
`media="media query"`	Query to determine applicable media
`type="media type"`	Media (MIME) type of embedded image file
`width="number"`	Horizontal dimension when used within **`picture`**
`height="number"`	Vertical dimension when used within **`picture`**
`svg`	Adds an inline SVG image
`x="number"`	The horizontal coordinate that positions the element in its viewport
`y="number"`	The vertical coordinate that positions the element in its viewport
`width="number"`	The width of the image in user units (1 user unit = 1 pixel by default)
`height="number"`	The height of the image in user units (1 user unit = 1 pixel by default)
	The **`svg`** *element has many additional attributes related to SVG functionality. See* ***w3.org/TR/SVG2/struct.html#SVGElement*** *for details.*

CHAPTER

8

TABLE MARKUP

IN THIS CHAPTER

How tables are used

Basic table structure

Spanning rows and columns

Row and column groups

Making tables accessible

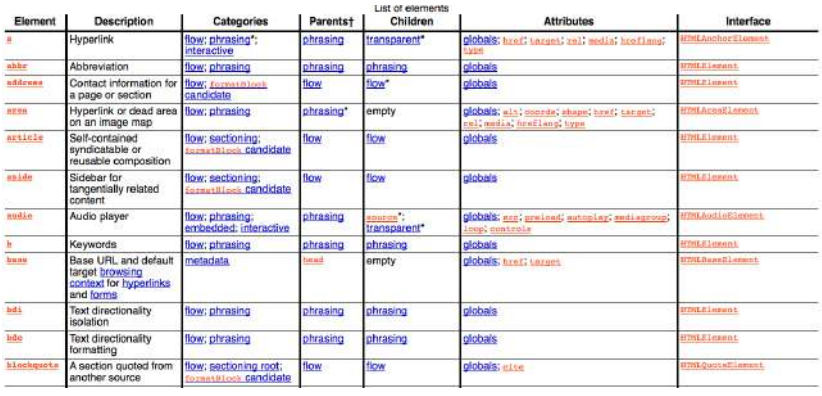
Before we launch into the markup for tables, let's check on our progress so far. We've covered a lot of territory: how to establish the basic structure of an HTML document, how to mark up text to give it meaning and structure, how to make links, and how to embed simple images in a page.

This chapter and the next two chapters, **Chapter 9, Forms**, and **Chapter 10, Embedded Media**, describe the markup for specialized content that you might not have a need for right away. If you're getting antsy to make your pages look good, you can skip right to **Part III, CSS for Presentation,** and start playing with styles. These three chapters will be waiting for you when you're ready for them.

Are you still here? Great. Let's talk tables. We'll start out by reviewing how tables should be used, then learn about the elements used to create them. Remember, this is an HTML chapter, so we're going to focus on the markup that structures the table content, and we won't be concerned with how the tables look (that will be tackled in various CSS chapters in **Part III**).

HOW TO USE TABLES

HTML tables were created for instances when you need to add **tabular material** (data arranged into rows and columns) to a web page. Tables may be used to organize schedules, product comparisons, statistics, or other types of information, as shown in FIGURE 8-1. Note that "data" doesn't necessarily mean numbers. A table cell may contain any sort of information, including numbers, text elements, and even images and multimedia objects.



Element	Description	Categories	Parents†	List of elements Children	Attributes	Interface
a	Hyperlink	flow; phrasing*; interactive	phrasing	transparent*	globals; href; target; rel; media; hreflang; type	HTMLAnchorElement
abbr	Abbreviation	flow; phrasing	phrasing	phrasing	globals	HTMLElement
address	Contact information for a page or section	flow; formatBlock candidate	flow	flow*	globals	HTMLElement
area	Hyperlink or dead area on an image map	flow; phrasing	phrasing*	empty	globals; alt; coords; shape; href; target; rel; media; hreflang; type	HTMLAreaElement
article	Self-contained syndicatable or reusable composition	flow; sectioning; formatBlock candidate	flow	flow	globals	HTMLElement
aside	Sidebar for tangentially related content	flow; sectioning; formatBlock candidate	flow	flow	globals	HTMLElement
audio	Audio player	flow; phrasing; embedded; interactive	phrasing	source*; transparent*	globals; src; preload; autoplay; mediagroup; loop; controls	HTMLAudioElement
b	Keywords	flow; phrasing	phrasing	phrasing	globals	HTMLElement
base	Base URL and default target browsing context for hyperlinks and forms	metadata	head	empty	globals; href; target	HTMLBaseElement
bdi	Text directionality isolation	flow; phrasing	phrasing	phrasing	globals	HTMLElement
bdo	Text directionality formatting	flow; phrasing	phrasing	phrasing	globals	HTMLElement
blockquote	A section quoted from another source	flow; sectioning root; formatBlock candidate	flow	flow	globals; cite	HTMLQuoteElement

w3c.org

PM	7:30	8:00	8:30	9:00	9:30	10:00	10:30
ABC	*The Adventures of Ozzie and Harriet*	The Patty Duke Show	*Gidget*	*The Big Valley*		Amos Burke — Secret Agent*	
CBS	Lost in Space		*The Beverly Hillbillies* #8 25.9 rating	*Green Acres* #11 24.6 rating	The Dick Van Dyke Show #16 23.6 rating	*The Danny Kaye Show*	
NBC	The Virginian #25 22.0 rating			*Bob Hope Presents the Chrysler Theatre / Chrysler Presents a Bob Hope Special*		*I Spy*	

wikipedia.org

mbta.org

FIGURE 8-1. Examples of tables used for tabular information, such as charts, calendars, and schedules

In visual browsers, the arrangement of data in rows and columns gives readers an instant understanding of the relationships between data cells and their respective header labels. Bear in mind when you are creating tables, however, that some readers will be hearing your data read aloud with a screen reader or reading Braille output. Later in this chapter, we'll discuss measures you can take to make table content accessible to users who don't have the benefit of visual presentation.

In the days before style sheets, tables were the only option for creating multicolumn layouts or controlling alignment and whitespace. Layout tables, particularly the complex nested table arrangements that were once standard web design fare, have gone the way of the dodo. If you need rows and columns for presentation purposes, there are alternatives that use CSS to achieve the

desired effect such as Flexbox (see **Chapter 17, Get In Line with Flexbox**) and CSS Grid Layout (see **Chapter 18, CSS Grid Layout**).

This chapter focuses on HTML table elements used to semantically mark up rows and columns of data as described in the HTML specification.

MINIMAL TABLE STRUCTURE

Let's take a look at a simple table to see what it's made of. Here is a small table with three rows and three columns that lists nutritional information:

Menu item	Calories	Fat (g)
Chicken noodle soup	120	2
Caesar salad	400	26

`<table>...</table>`
Tabular content (rows and columns)

`<tr>...</tr>`
Table row

`<th>...</th>`
Table header

`<td>...</td>`
Table cell data

FIGURE 8-2 reveals the structure of this table according to the HTML table model. All of the table's content goes into cells that are arranged into rows. Cells contain either header information (titles for the columns, such as "Calories") or data, which may be any sort of content.

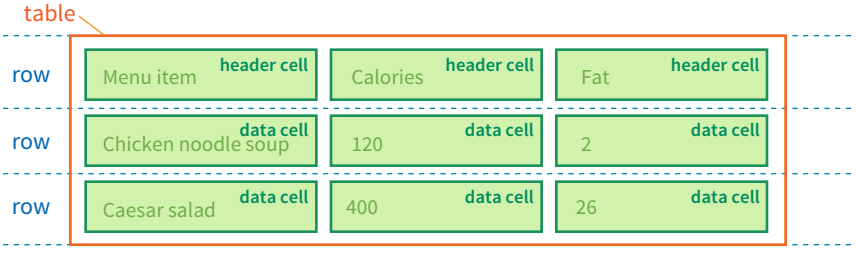


FIGURE 8-2. Tables are made up of rows that contain cells. Cells are the containers for content.

Simple enough, right? Now let's look at how those parts translate into elements (FIGURE 8-3).

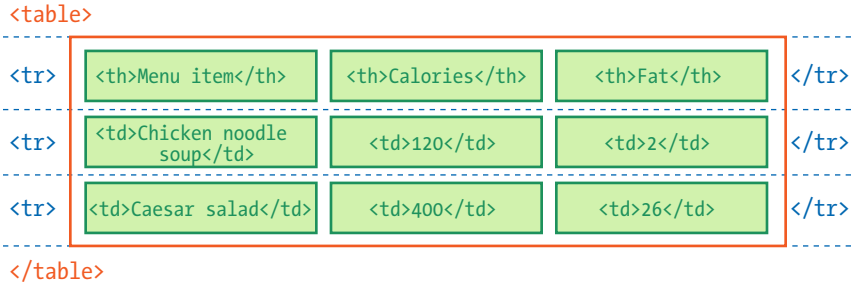


FIGURE 8-3. The elements that make up the basic structure of a table

Styling Tables

Once you build the structure of the table in the markup, you can add a layer of style to customize its appearance. Style sheets can and should be used to control these aspects of a table's visual presentation. We'll get to all the formatting tools you'll need in the following chapters—here are a few pointers to the relevant topics:

Chapter 5, Formatting Text:

- Font settings for cell contents
- Text color in cells

Chapter 13, Colors and Backgrounds:

- Background colors
- Tiling background images

Chapter 15, Thinking Inside the Box (CSS Box Model):

- Table dimensions (width and height)
- Borders
- Cell padding (space around cell contents)
- Margins around the table

Chapter 21, More CSS Techniques:

- Special properties for controlling borders and spacing between cells

FUN WITH THE SPEC

According to the HTML spec, a `table` element may contain "in this order: optionally a `caption` element, followed by zero or more `colgroup` elements, followed optionally by a `thead` element, followed by either zero or more `tbody` elements or one or more `tr` elements, followed optionally by a `tfoot` element (but there can only be one `tfoot` element child in total)."

Well, glad we cleared that up!

FIGURE 8-3 shows the elements that identify the table (`table`), rows (`tr`, for "table row"), and cells (`th`, for "table headers," and `td`, for "table data"). Cells are the heart of the table, because that's where the actual content goes. The other elements just hold things together.

What we don't see are column elements. The number of columns in a table is implied by the number of cells in each row. This is one of the things that make HTML tables potentially tricky. Rows are easy—if you want the table to have three rows, just use three `tr` elements. Columns are different. For a table with four columns, you need to make sure that every row has four `td` or `th` elements. (There's more to the column story, which I'll cover in the section **"Row and Column Groups"** later in this chapter.)

Written out in a source document, the markup for the table in FIGURE 8-3 looks like the following sample. It's common to stack the `th` and `td` elements in order to make them easier to find in the source, as shown here; this does not affect how the browser renders them:

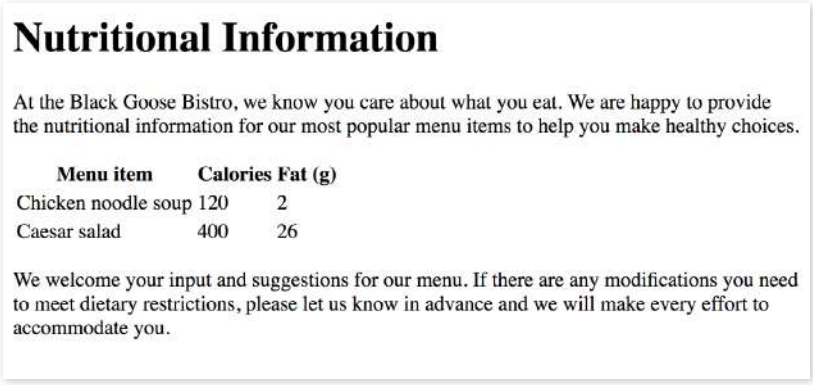
```
<table>
   <tr>
      <th>Menu item</th>
      <th>Calories</th>
      <th>Fat (g)</th>
   </tr>
   <tr>
      <td>Chicken noodle soup</td>
      <td>120</td>
      <td>2</td>
   </tr>
   <tr>
      <td>Caesar salad</td>
      <td>400</td>
      <td>26</td>
   </tr>
</table>
```

Remember, all the content must go in cells—that is, within `td` or `th` elements. You can put any content in a cell: text, a graphic, or even another table.

Start and end `<table>` tags identify the beginning and end of the tabular material. The `table` element may directly contain only some number of `tr` (row) elements, a caption, and, optionally, the row and column group elements listed in the **"Row and Column Groups"** section. The only thing that can go in the `tr` element is some number of `td` and/or `th` elements. In other words, there may be no text content within the `table` and `tr` elements that isn't contained within a `td` or `th`.

Finally, FIGURE 8-4 shows how this table would look in a simple web page, as displayed using the browser's default styles. I know it's not exciting. What is worth noting is that tables always start on new lines by default in browsers.

Here is the source for another table. Can you tell how many rows and columns it will have when it is displayed in a browser?



Nutritional Information

At the Black Goose Bistro, we know you care about what you eat. We are happy to provide the nutritional information for our most popular menu items to help you make healthy choices.

Menu item	Calories	Fat (g)
Chicken noodle soup	120	2
Caesar salad	400	26

We welcome your input and suggestions for our menu. If there are any modifications you need to meet dietary restrictions, please let us know in advance and we will make every effort to accommodate you.

FIGURE 8-4. The default rendering of the sample table in a browser

```
<table>
  <tr>
    <th>Burgers</th>
    <td>Organic Grass-fed Beef</td>
    <td>Black Bean Veggie</td>
  </tr>
     <tr>
    <th>Fries</th>
    <td>Hand-cut Idaho potato</td>
    <td>Seasoned sweet potato</td>
   </tr>
</table>
```

If you guessed that it's a table with two rows and three columns, you are correct! Two **tr** elements create two rows; one **th** and two **td** elements in each row create three columns.

TABLE HEADERS

As you can see in FIGURE 8-4, the text marked up as headers (**th** elements) displays differently by default from the other cells in the table (**td** elements). The difference, however, is not purely cosmetic. Table headers are important because they provide information or context about the cells in the row or column they precede. The **th** element may be handled differently than **td**s by alternative devices. For example, screen readers may read the header aloud before each data cell ("Menu item: Caesar salad, Calories: 400, Fat-g: 26").

In this way, headers are a key tool for making table content accessible. Don't try to fake them by formatting a row of **td** elements differently than the rest of the table. Conversely, don't avoid using **th** elements because of their default rendering (bold and centered). Instead, mark up the headers semantically and change the presentation later with a style rule.

That covers the basics. Before we get fancier, try your hand at EXERCISE 8-1.

EXERCISE 8-1.
Making a simple table

Try writing the markup for the table shown in FIGURE 8-5. You can open a text editor or just write it down on paper. The finished markup is provided in the *materials* folder (*learningwebdesign.com/6e/materials*).

Note that I've added a 1-pixel border around cells with a style rule just to make the structure clear. If you would like borders on your tables, copy this **style** element into the **head** of the document(s) you create for the exercises in this chapter:

```
<style>
  td, th {
    border: 1px solid gray;
  }
</style>
```

Take the time to close all table elements with closing tabs (for example, </tr>). Technically, you are not *required* to close **tr**, **th**, and **td** elements, but it is a strong convention, and in my opinion, it makes the markup tidier and easier to parse.

Album	Year
Rubber Soul	1965
Revolver	1966
Sgt. Pepper's	1967
The White Album	1968
Abbey Road	1969

FIGURE 8-5. Write the markup for this table.

EXERCISE 8-2. Column spans

Try writing the markup for the table shown in FIGURE 8-7. You can open a text or code editor or just write it down on paper. I've added borders to reveal the cell structure in the figure, but your table won't have them unless you add the style rule shown in EXERCISE 8-1. Again, the final markup is provided in the *materials* folder.

Some hints:

- The first and third rows show that the table has a total of three columns.
- When a cell is spanned over, its `td` element does not appear in the table.

7:00pm	7:30pm	8:00pm
The Sunday Night Movie		
Perry Mason	Candid Camera	What's My Line?
Bonanza	The Wackiest Ship in the Army	

FIGURE 8-7. Practice column spans by writing the markup for this table.

SPANNING CELLS

One fundamental feature of table structure is cell **spanning**, which is the stretching of a cell to cover several rows or columns. Spanning cells allows you to create complex table structures, but it has the side effect of making the markup a little more difficult to keep track of. It can also potentially make it more difficult for users with screen readers to follow.

You make a header or data cell span by adding the `colspan` or `rowspan` attributes, as I'll discuss next.

Column Spans

Column spans, created with the `colspan` attribute in the `td` or `th` element, stretch a cell to the right to span over the subsequent column or columns (FIGURE 8-6). Here, a column span is used to make a header apply to two columns (a border added around the cells with CSS reveals the structure of the table in the screenshot):

```
<table>
  <tr>
    <th colspan="2">Fat</th>
  </tr>
  <tr>
    <td>Saturated Fat (g)</td>
    <td>Unsaturated Fat (g)</td>
  </tr>
</table>
```

Fat	
Saturated Fat (g)	Unsaturated Fat (g)

FIGURE 8-6. The `colspan` attribute stretches a cell to the right to span the specified number of columns.

Notice in the first row (`tr`) that there is only one `th` element, while the second row has two `td` elements. The `th` for the column that was spanned over is no longer in the source; the cell with the `colspan` stands in for it. Every row should have the same number of cells or equivalent `colspan` values. For example, there are two `td` elements and the `colspan` value is 2, so the implied number of columns in each row is equal (see **Warning**).

Try your hand at column spanning in EXERCISE 8-2.

WARNING

Be careful with `colspan` *values. If you specify a number that exceeds the number of columns in the table, browsers add columns to the existing table, which may be problematic.*

Row Spans

Row spans, created with the **rowspan** attribute, work just like column spans, but they cause the cell to span downward over two or more rows. In this example, the first cell in the table spans down three rows (FIGURE 8-8):

```
<table>
  <tr>
    <th rowspan="3">Serving Size</th>
    <td>Small (8oz.)</td>
  </tr>
  <tr>
    <td>Medium (16oz.)</td>
  </tr>
  <tr>
    <td>Large (24oz.)</td>
  </tr>
</table>
```

Serving Size	Small (8oz.)
	Medium (16oz.)
	Large (24oz.)

FIGURE 8-8. The **rowspan** attribute stretches a cell downward to span the specified number of rows.

Again, notice that the **td** elements for the cells that were spanned over (the first cells in the remaining rows) do not appear in the source. The **rowspan="3"** implies cells for the subsequent two rows, so no **td** elements are needed.

If you enjoyed spanning columns, you'll love spanning rows in EXERCISE 8-3.

Space in and Between Cells

By default, tables expand just enough to fit the content of the cells, which can look a little cramped. Old versions of HTML included **cellpadding** and **cellspacing** attributes for adding space within and between cells, but they have been kicked out of the current HTML spec as they are obsolete, presentational markup. The proper way to adjust table cell spacing is with style sheets, of course. The article **"Styling Tables"** available at *learningwebdesign.com/articles* has all the details once you are more familiar with CSS.

TABLE ACCESSIBILITY

As a web designer, it is important that you always keep in mind how your site's content is going to be used by visitors with impaired sight. It's especially challenging to make sense of tabular material when using a screen reader, but the HTML specification provides measures to improve the experience and make your content more understandable.

EXERCISE 8-3.
Row spans

Try writing the markup for the table shown in FIGURE 8-9. Remember that cells that are spanned over do not appear in the table code.

Some hints:

- Rows always span downward, so the "oranges" cell is part of the first row even though its content is vertically centered.
- Cells that are spanned over do not appear in the code.

apples	oranges	pears
bananas		pineapple
lychees		

FIGURE 8-9. Practice row spans by writing the markup for this table.

Describing Table Content

<caption>...</caption>
Title or description to be displayed with the table

The most basic way to give sight-impaired users an overview of your table is to give it a title or description with the **caption** element. Captions generally display above to the table and can be used to describe the table's contents or provide hints on how it is structured.

When used, the **caption** element must be the first thing within the **table** element, as shown in this example, which adds a caption to the nutritional chart from earlier in the chapter:

```
<table>
  <caption>Nutritional Information</caption>
  <tr>
    <th>Menu item</th>
    <th>Calories</th>
    <th>Fat (g)</th>
  </tr>
  <!-- table continues -->
</table>
```

The caption is displayed above the table by default, as shown in FIGURE 8-10, although you can use a style sheet property to move it below the table (**caption-side: bottom**).

Nutritional Information

Menu item	Calories	Fat (g)
Chicken noodle soup	120	2
Caesar salad	400	26

FIGURE 8-10. The table caption is displayed above the table by default.

For longer descriptions, you could consider putting the table in a **figure** element and using the **figcaption** element for the description. The HTML specification has a number of suggestions for providing table descriptions (*oreil.ly/aWN0l*).

Connecting Cells and Headers

We discussed headers briefly as a straightforward method for improving the accessibility of table content, but sometimes it may be difficult to know which header applies to which cells. For example, headers may be at the left or right edge of a row rather than at the top of a column. And although it may be easy for sighted users to understand a table structure at a glance, for users hearing the data as text, the overall organization is not as clear. The **scope** and **headers** attributes allow authors to explicitly associate headers and their respective content:

`scope`

The **scope** attribute associates a table header with the row, column, group of rows (such as **tbody**), or column group in which it appears by using the values **row**, **col**, **rowgroup**, or **colgroup**, respectively. This example uses the **scope** attribute to declare that a header cell applies to the current row:

```
<tr>
  <th scope="row">Mars</th>
  <td>.95</td>
  <td>.62</td>
  <td>0</td>
</tr>
```

Accessibility experts recommend that every **th** element contain a **scope** attribute to make its associated data explicitly clear.

`headers`

For complicated tables in which **scope** is not sufficient to associate a table data cell with its respective header (such as when the table contains multiple spanned cells), the **headers** attribute is used in the **td** element to explicitly tie it to a header's **id** value. In this example, the cell content ".38" is tied to the header "Diameter measured in earths":

```
<th id="diameter">Diameter measured in earths</th>
<!-- many other cells -->
<td headers="diameter">.38</td>
<!-- many other cells -->
```

While this method does link headers to cells more explicitly, it requires more markup, is prone to errors, and is not universally supported by screen readers (even though it is supported by current browsers). The recommended best practice is to create tables in such a way that a simple **scope** attribute will do the job.

This section obviously covers only the tip of the iceberg of table accessibility. In-depth instruction on authoring accessible tables is beyond the scope of this beginner book. If you'd like to learn more, the WebAIM article "Creating Accessible Tables" (*webaim.org/techniques/tables/data*) is an excellent starting point.

There is one more important set of elements for helping make the semantic structure of a table clear: row and column grouping elements.

ROW AND COLUMN GROUPS

The sample tables we've been looking at so far in this chapter have been stripped down to their bare essentials to make the structure clear while you're learning how tables work. But tables in the real world are not always so simple. Check out the beauty in FIGURE 8-11, from the CSS Writing Modes Level 3 spec. You can see in the example three groups of columns (one with

headers, two with two columns each) and three groupings of rows (headers, data, and a footnote).

Conceptual table groupings like these are marked up with row group and column group elements that provide additional semantic structure and more "hooks" for styling or scripting. For example, the row and column groups in FIGURE 8-11 were styled with thicker borders to make them stand out visually.

VIEW SOURCE

You can view the source of the table in FIGURE 8-11 at *w3.org/TR/css-writing-modes-3/#unicode-bidi*. It's too long to print here, but it is clearly marked up and easy to follow. Note that it uses all the row group elements, column groups, and the **scope** attribute we saw in the last section to associate headers with rows. There are several interesting tables on this page for your source-viewing pleasure.

Bidi control codes injected by 'unicode-bidi' at the start/end of 'display: inline' boxes

'unicode-bidi' value	'direction' value: 'ltr' start	'ltr' end	'rtl' start	'rtl' end
'normal'	–	–	–	–
'embed'	LRE (U+202A)	PDF (U+202C)	RLE (U+202B)	PDF (U+202C)
'isolate'	LRI (U+2066)	PDI (U+2069)	RLI (U+2067)	PDI (U+2069)
'bidi-override'*	LRO (U+202D)	PDF (U+202C)	RLO (U+202E)	PDF (U+202C)
'isolate-override'*	FSI,LRO (U+2068,U+202D)	PDF,PDI (U+202C,U+2069)	FSI,RLO (U+2068,U+202E)	PDF,PDI (U+202C,U+2069)
'plaintext'	FSI (U+2068)	PDI (U+2069)	FSI (U+2068)	PDI (U+2069)

* The LRO/RLO+PDF pairs are also applied to the root inline box of a block container if these values of 'unicode-bidi' were specified on the block container.

FIGURE 8-11. An example of a table with row and column groups (from the CSS Writing Modes Level 3 specification)

Row Group Elements

`<thead>...</thead>`
Table header row group

`<tbody>...</tbody>`
Table body row group

`<tfoot>...</tfoot>`
Table footer row group

You can describe rows or groups of rows as belonging to a header, footer, or the body of a table by using the **thead**, **tfoot**, and **tbody** elements, respectively.

Authors may use these elements to apply styles to various regions of a table, but they could also be used to improve the usability of long tables. For example, some user agents (another word for a browser) may attach a header row to the top edge of the window and allow the table content to scroll below it. Similarly, a header and footer could be repeated on every page when the document is printed. Row group elements may only contain one or more **tr** elements. They contain no direct text content. The **thead** element should appear first, followed by any number of **tbody** elements, followed by an optional **tfoot**. It's worth noting that the **tbody** element is implied and always there in the DOM, whether you mark it up explicitly or not.

This is the row group markup for the table in FIGURE 8-11 (**td** and **th** elements are hidden to save space):

```
<table>
...
<thead>
  <!-- headers in these rows-->
  <tr></tr>
  <tr></tr>
  <tr></tr>
<thead>
<tbody>
  <!-- data -->
  <tr></tr>
```

```
    <tr></tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
  </tbody>
  <tfoot>
    <!-- footnote -->
    <tr></tr>
  </tfoot>
  </table>
```

Column Group Elements

As you've learned, columns are implied by the number of cells (**td** or **th** elements) in each row. You can semantically group columns (and assign **id** and **class** values) using the **colgroup** element.

`<colgroup>...</colgroup>`
A semantically related group of columns

`<col>...</col>`
One column in a column group

Column groups are identified at the start of the table, just after the **caption**, if there is one, and they give the browser a little heads-up as to the column arrangement in the table. The number of columns a **colgroup** represents is specified with the **span** attribute. Here is the column group section at the beginning of the table in FIGURE 8-11:

```
<table>
  <caption>...</caption>
  <colgroup></colgroup>
  <colgroup span="2"></colgroup>
  <colgroup span="2"></colgroup>
  <!-- rest of table -->
```

That's all there is to it. If you need to access individual columns within a **colgroup** for scripting or styling, identify them with **col** elements. The previous column group section could also have been written like this:

```
<colgroup></colgroup>
<colgroup>
  <col class="start">
  <col class="end">
</colgroup>
<colgroup>
  <col class="start">
  <col class="end">
</colgroup>
```

Note that the **colgroup** elements contain no content—they only provide an indication of semantically relevant column structure. The **col** elements are useful as handles for scripts or styles but are not required. Note that they are void and do not have content.

NOTE

When **colgroup** *elements contain* **col** *elements, they must not have a* **span** *attribute.*

WRAPPING UP TABLES

This chapter gave you a good overview of the components of HTML tables. EXERCISE 8-4 combines most of what we've covered to give you a little more practice at authoring tables.

EXERCISE 8-4. The table challenge

Now it's time to put together the table-writing skills you've acquired in this chapter. Your challenge is to write out the source document for the table shown in FIGURE 8-12.

I'll walk you through it one step at a time:

1. First, open a new document in your text editor and set up its overall structure (**DOCTYPE**, **html**, **head**, **title**, and **body** elements). Save the document as *table.html* in the directory of your choice.
2. Next, in order to make the boundaries of the cells and table clear when you check your work, I'm going to have you add some simple style sheet rules to the document. Don't worry about understanding exactly what's happening here (although it's fairly intuitive). Insert this **style** element in the **head** of the document exactly as you see it here:

```
<head>
  <title>Table Challenge</title>
  <style>
    td, th { border: 1px solid #CCC; }
    table { border: 1px solid black; }
  </style>
</head>
```

3. Now it's time to start building the table. It helps to start by setting up the table and adding as many empty row elements as you'll need for the final table as placeholders, as shown here. You can tell from the figure that there are five rows in this table:

```
<body>
  <table>
    <tr></tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
  </table>
</body>
```

4. Start with the top row, and fill in the **th** and **td** elements from left to right, including any row or column spans, as necessary. I'll help with the first row.

   The first cell (the one in the top-left corner) spans down the height of two rows, so it gets a **rowspan** attribute. We'll use a **th** here to keep it consistent with the rest of the row. This cell has no content:

```
<table>
  <tr>
    <th rowspan="2"></th>
  </tr>
```

   The cell in the second column of the first row spans over the width of two columns, so it gets a **colspan** attribute:

```
<table>
  <tr>
    <th rowspan="2"></th>
    <th colspan="2">A common header for two
subheads</th>
  </tr>
```

   The cell in the third column has been spanned over by the **colspan** we just added, so we don't need to include it in the markup. The cell in the fourth column also spans down two rows:

```
<table>
  <tr>
    <th rowspan="2"></th>
    <th colspan="2">A common header for two
subheads</th>
    <th rowspan="2">Header 3</th>
  </tr>
```

5. Now it's your turn. Continue filling in the **th** and **td** elements for the remaining four rows of the table. Here's a hint: the first and last cells in the second row have been spanned over. Also, if it's bold in the example, make it a header.
6. To complete the content, add the title over the table by using the **caption** element.
7. Use the **scope** attribute to make sure that the "Thing A," "Thing B," and "Thing C" headers are associated with their respective rows.
8. Finally, give the table row and column groups for greater sematic clarity. There is no **tfoot** in this table. There are two column groups: one column for headers, the rest for data. Use the **span** attribute (no need for individual column identification).
9. Save your work and open the file in a browser. The table should look just like the one on this page. If not, go back and adjust your markup. If you're stumped, the final markup for this exercise is provided in the *materials* folder.

Your Content Here

	A common header for two subheads		Header 3
	Header 1	Header 2	
Thing A	data A1	data A2	data A3
Thing B	data B1	data B2	data B3
Thing C	data C1	data C2	data C3

FIGURE 8-12. The table challenge

TEST YOURSELF

The answers to these questions appear in **Appendix A**.

1. What are the parts (elements) of a basic HTML table?

2. What elements can a **`table`** contain directly (i.e., first-level children)?

3. What elements can a **`tr`** contain?

4. When would you use the **`colgroup`** (column) and **`col`** (column) elements?

5. How could you improve the accessibility of the following table?

```
<table>
  <tr>
    <th>Username</th>
    <th>First Name</th>
    <th>Last Name</th>
  </tr>
  <tr>
    <td>bobiscool</td>
    <td>Bob</td>
    <td>Stevenson</td>
  </tr>
  <tr>
    <td>soisjane</td>
    <td>Jane</td>
    <td>Smith</td>
  </tr>
</table>
```

6. Find four errors in this table markup:

```
<caption>Primetime Television 1965</caption>
<table>
  Thursday Night
  <tr></tr>
    <th>7:30</th>
    <th>8:00</th>
    <th>8:30</th>
  <tr>
    <td>Shindig</td>
    <td>Donna Reed Show</td>
    <td>Bewitched</td>
  </tr>
  <tr>
    <colspan="2">Laredo</colspan>
    <td>Daniel Boone</td>
  </tr>
</table>
```

ELEMENT REVIEW: TABLES

The following is a summary of the elements we covered in this chapter.

Element and attributes	Description
`table`	Establishes a table element
`tr`	Establishes a row within a table
`td`	Establishes a cell within a table row
`colspan="number"`	Number of columns the cell should span
`rowspan="number"`	Number of rows the cell should span
`headers="header name"`	Associates the data cell with a header
`th`	Table header associated with a row or column
`abbr="text"`	Alternative label for when the header cell is referenced in other contexts
`colspan="number"`	Number of columns the cell should span
`rowspan="number"`	Number of rows the cell should span
`headers="header name"`	Associates a header with another header
`scope="row|col|rowgroup|colgroup"`	Associates the header with a row, row group, column, or column group
`caption`	Gives the table a title that displays in the browser
`colgroup`	Declares a group of columns
`span="number"`	Number of columns the column group spans; may not be used when the **`colgroup`** contains **`col`** elements
`col`	Declares a column
`span="number"`	Number of columns the column spans
`tbody`	Identifies a table body row group
`thead`	Identifies a table header row group
`tfoot`	Identifies a table footer row group

CHAPTER 9

FORMS

IN THIS CHAPTER

How forms work

Elements for adding form widgets

Making forms accessible

Form design basics

It didn't take long for the web to shift from a network of pages to read to a place where you go to get things *done*—making purchases, booking plane tickets, signing petitions, searching a site, posting to social media...the list goes on! Web forms handle all of these interactions. We have a large collection of widgets and form controls that make it easy for users to fill out forms and for developers to create them. As HTML evolved, it became possible for tasks that once relied on JavaScript to be handled by markup and native browser behavior alone.

This chapter introduces web forms, how they work, all those widgets, and the markup used to create them. I'll also briefly discuss the importance of web form design.

HOW FORMS WORK

There are two parts to a working form. The first part is the form that you see on the page itself that is created using HTML markup. Forms are made up of buttons, input fields, and drop-down menus (collectively known as **form controls** or **widgets**) used to collect information from the user. Forms may also contain text and other elements.

The other component of a web form is an application or script on the server that processes the information collected by the form and returns an appropriate response. It's what makes the form *work*. In other words, posting an HTML document with form elements isn't enough. Web applications and scripts require programming know-how that is beyond the scope of this book, but the **"Getting Your Forms to Work"** sidebar later in this chapter provides some options for getting the scripts you need.

From Data Entry to Response

If you are going to be creating web forms, it is beneficial to understand what is happening behind the scenes. This example traces the steps of a transaction using a simple form that gathers names and email addresses for a mailing list; however, it is typical of the process for many forms:

1. Your visitor—let's call her Sally—opens the page with a web form in the browser window. The browser sees the form control elements in the markup and renders them with the appropriate form controls on the page, including two text entry fields and a submit button (shown in FIGURE 9-1).
2. Sally would like to sign up for this mailing list, so she enters her name and email address into the fields and submits the form by hitting the submit button.
3. The browser collects the information she entered, encodes it (see the sidebar **"A Word About Encoding"**), and sends it to the web application on the server.
4. The web application accepts the information and processes it (that is, does whatever it is programmed to do with it). In this example, the name and email address are added to a mailing list database.
5. The web application also returns a response. The kind of response sent back depends on the content and purpose of the form. Here, the response is a simple web page saying thank you for signing up for the mailing list. Other applications might respond by reloading the form page with updated information, by moving the user to another related form page, or by issuing an error message if the form is not filled out correctly, to name only a few examples.
6. The server sends the web application's response back to the browser, where it is displayed. Sally can see that the form worked and that she has been added to the mailing list.

A Word About Encoding

Form data is encoded via the same method used for URLs. Spaces and other characters that are not permitted get translated into their hexadecimal equivalents. For example, each space character in the collected form data is represented by the character **+** or **%20**, and a slash (**/**) character is replaced with **%2F**. You don't need to worry about this; the browser handles it automatically.

JavaScript for Forms

So far we've talked about server-side programs that process form information, and we're going to spend the rest of the chapter reviewing the form elements that go on the page, but I don't want to minimize the importance of JavaScript in modern form design and functionality. JavaScript is used in many ways that improve the experience using the form, such as:

- Giving users feedback as they type the form, for example, indicating the strength of a new password as they type it in.
- Prefilling parts of the form automatically based on other info, for example, filling in the city and state when provided with the postal code.
- Guiding how users enter data, like automaticaly adding dashes in a phone number or slashes in a date.
- Hiding or revealing fields or inputs based on user input, such as showing a different set of form inputs based on a selection.
- Validating the form on the client side before it is sent to the server, including making sure the required fields are filled and in the correct format. For example, a script could be used to make sure that an email address input is filled and the text has the expected structure of an email address.
- Saving what users type in as they fill it out so it is available if they are somehow interrupted

Unfortunately, form scripting is beyond the scope of this book; however, you will get a solid introduction to JavaScript as well as some hands-on experience in **Part IV: JavaScript for Behavior**.

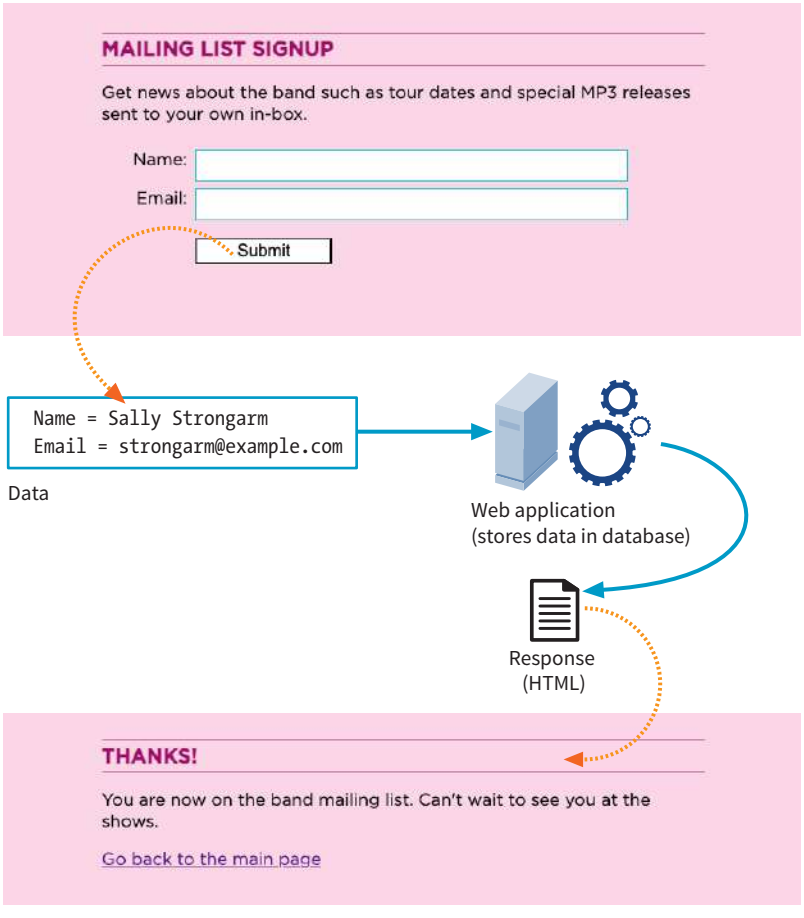

MAILING LIST SIGNUP
Get news about the band such as tour dates and special MP3 releases sent to your own in-box.
Name:
Email:
Submit
Name = Sally Strongarm
Email = strongarm@example.com
Data
Web application
(stores data in database)
Response
(HTML)
THANKS!
You are now on the band mailing list. Can't wait to see you at the shows.
Go back to the main page

FIGURE 9-1. What happens behind the scenes when a web form is submitted

THE FORM ELEMENT

Forms are added to web pages with (no surprise here) the `form` element. The `form` element is a container for all the content of the form, including some number of form controls, such as text entry fields and buttons. It may also contain block elements (`h1`, `p`, and lists, for example). However, it may *not* contain another `form` element.

`<form>...</form>`
Interactive form

This sample source document contains a form similar to the one shown in FIGURE 9-1:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Mailing List Signup</title>
  <meta charset="utf-8">
</head>
<body>
  <h1>Mailing List Signup</h1>
```

■ MARKUP TIP

Be careful not to nest `form` elements or allow them to overlap. A `form` element must be closed before the next one begins.

```
  <form action="/mailinglist.php" method="post">
    <fieldset>
      <legend>Join our email list</legend>
      <p>Get news about the band such as tour dates and special MP3
releases sent to your own in-box.</p>
      <ol>
        <li><label for="firstlast">Name:</label>
          <input type="text" name="fullname" id="firstlast"></li>
        <li><label for="email">Email:</label>
          <input type="text" name="email" id="email"></li>
      </ol>
      <input type="submit" value="Submit">
    </fieldset>
  </form>

</body>
</html>
```

NOTE

It is current best practice to wrap form controls in semantic HTML elements such as lists, paragraphs, or `div`*s. Ordered lists, as shown in this example, are a popular solution, but know that there are often default styles that you'll need to clear out before styling them, particularly on mobile browsers. The* `fieldset`*,* `legend`*, and* `label` *elements used in the example improve accessibility. They will be explained later in this chapter.*

In addition to being a container for form control elements (see **Note**), the `form` element has some attributes that are necessary for interacting with the form processing program on the server. Let's take a look at each.

The action Attribute

The `action` attribute provides the location (URL) of the application or script that will be used to process the form. The `action` attribute in this example sends the data to a script on the server called *mailinglist.php*:

```
<form action="/mailinglist.php" method="post">...</form>
```

The *.php* suffix indicates that this form is processed by a script written in the PHP scripting language, but web forms may be processed by any of the following technologies (as well others not listed here):

- **PHP**, an open source scripting language most commonly used with the Apache web server. This is one of the most popular and widely supported forms processing options.
- **Node.js**, a JavaScript runtime that has gained popularity with the rise of JavaScript on both the client and server sides. Developers can use JavaScript for form processing on the server side using frameworks like Express.js.
- **Python**, a popular scripting language for web and server applications. With frameworks like Django and Flask, it is widely used for web development, including form processing.
- **Ruby on Rails**, a development platform that uses the Ruby programming language. Many popular web applications are built with it.
- **Java**, a long-time staple for server-side development. Java-based frameworks like Spring are commonly used for handling web forms and processing data.

- **Microsoft's ASP.NET**, which uses C# as its primary language. ASP.NET is popular for web development on the Windows platform and includes features for handling web forms.
- **Go**, which has gained popularity for its simplicity and performance. Also called Golang, it is used for server-side development, and there are frameworks like Gin that facilitate web form processing.

Form processing options may have their own suffixes, such as *.php* or ASP.NET's *.aspx*, or none at all (as is the case for the Ruby on Rails platform). Check with your developer, server administrator, or script documentation for the proper name and location of the program to be provided by the **`action`** attribute.

Sometimes there is form processing code such as PHP embedded right in the HTML file. In that case, leave the **`action`** empty, and the form will post to the page itself.

The method Attribute

The **`method`** attribute

specifies how the information should be sent to the server. Let's use this data gathered from the sample form in FIGURE 9-1 as an example:

```
fullname = Sally Strongarm
email = strongarm@example.com
```

When the browser encodes that information for its trip to the server, it looks like this (note that the character space and **`@`** symbol have been encoded as **`%20`** and **`%40`**, respectively, for transport):

```
fullname=Sally%20Strongarm&email=strongarm%40example.com
```

There are two methods for sending this encoded data via HTTP to the server: POST or GET (see **Note**), indicated by the **`method`** attribute in the **`form`** element. The method is optional and will default to GET if omitted. There is a third **`method`** keyword, **`dialog`**, which indicates the form is intended to close the dialog box it is in and not submit.

We'll look at the difference between POST and GET in the following sections. Our example uses the POST method, as shown here:

```
<form action="/mailinglist.php" method="post">...</form>
```

The POST method

When the form's method is set to POST, the browser sends a separate server request containing some special headers followed by the data. In theory, only the server sees the content of this request, and thus POST is the best method for sending secure information such as a home address or other personal information. In practice, make sure HTTPS (discussed in **Chapter 2, How the**

Getting Your Forms to Work

If you aren't a backend developer, don't fret. You have a few options for making your forms operational:

Use hosting plan goodies
: Many site hosting plans include access to scripts for simple functions such as mailing lists. More advanced plans may even provide everything you need to add a full shopping cart system to your site as part of your monthly hosting fee. Documentation or a technical support person should be available to help you use them.

Use a content management system
: Content management systems (CMSs) such as WordPress and Drupal have a lot of form functionality built in.

Hire a backend developer
: If you need a custom solution, you may need to hire a developer who has server-side programming skills. Tell them what you are looking to accomplish, and they will suggest a solution. Again, you need to make sure you have permission to install scripts on your server under your current hosting plan, and that the server supports the language you choose.

NOTE

You will often see the values POST and GET written in all capital letters. This is because that is how they appear in the HTTP protocol, and it became a convention to write them that way as attribute values as well. However, according to the current HTML spec, attribute values are not case-sensitive, so you can write **`POST`**, **`post`**, *or even* **`PosT`**, *and it will work the same. If you are writing documents with the stricter syntax of XHTML, the values must be all lowercase.*

Web Works) is enabled on your server so the user's data is encrypted and inaccessible in transit.

The POST method is also preferable for sending a lot of data, such as a lengthy text entry, because there is no character limit, as there is for GET.

In this chapter, I'll stick with the more prevalent POST method.

The GET method

With the default GET method, the encoded form data gets tacked right onto the URL sent to the server. A question mark character separates the URL from the following data, as shown here:

```
get http://www.bandname.com/mailinglist.php?name=Sally%20Strongarm&emai
l=strongarm%40example.com
```

The GET method is appropriate if you want users to be able to bookmark the results of a form submission (such as a list of search results). Because the content of the form is in plain sight, however, GET is not appropriate for forms requesting private personal or financial information. GET is also inappropriate if the form submission performs an action, such as deleting something or adding data to a database, because if the user hits the back button, it gets submitted again. In addition, GET may not be used when the form is used to upload a file.

Now that we've gotten through the technical aspects of the `form` element, let's turn our attention to form controls.

VARIABLES AND CONTENT

Web forms use a variety of controls that allow users to enter information or choose between options. Control types include various text entry fields, buttons, menus, and a few controls with special functions. They are added to the document with a collection of form control elements that we'll be examining one by one in the upcoming section **"The Great Form Control Roundup."**

As a web designer, you need to be familiar with the control options to make your forms easy and intuitive to use. It is also useful to have an idea of what form controls are doing behind the scenes.

The name Attribute

The job of each form control is to collect one bit of information from a user. In the previous form example, text entry fields collect the visitor's name and email address. To use the technical term, "fullname" and "email" are two **variables** collected by the form. The data entered in the control by the user (here, "Sally Strongarm" or "strongarm@example.com") becomes the **value** or content for its respective variable (see **Note**).

NOTE

Variables and values are a fundamental concept in programming languages as well. You'll see them in action in **Chapter 22, Getting Started with JavaScript**.

The **name** attribute provides the variable name for the control. In this example, the text gathered by a **textarea** element is defined as the value for the "comment" variable:

```
<textarea name="comment" rows="4" cols="45" placeholder="Leave us a
comment."></textarea>
```

When a user enters a comment in the field (e.g., "This is the best band ever!"), it will be passed to the server as a variable/value pair, like this:

```
comment=This%20is%20the%20best%20band%20ever%21
```

All form control elements must include a **name** attribute so the form processing application can sort the information. You may include a **name** attribute for submit and reset button elements, but they are not required, because they have special functions (submitting or resetting the form) not related to data collection.

All form controls (except submit and reset buttons) must include a name attribute.

Naming Your Variables

You can't just name controls willy-nilly. The web application that processes the data is programmed to look for specific variable names.

If the script or application will be created later, be sure to name your variables simply and descriptively and to document them well. In addition, to avoid confusion, you are advised to name each variable uniquely—that is, don't use the same name for two variables (however, there may be exceptions for which doing so is desirable). You should also avoid putting character spaces in variable names; use an underscore or hyphen instead.

If you are designing a form to work with a preexisting application or script, you need to find out the variable names to use so the inputs and the processing application are speaking the same language. You can get the variable names from the instructions provided with a ready-to-use script on your server, your system administrator, or the developer you are working with.

We've covered the basics of the **form** element and how form control variables are named. Now we can get to the real meat of form markup: the controls.

THE GREAT FORM CONTROL ROUNDUP

This is the fun part—playing with the markup that adds form controls to the page. This section introduces the elements used to create the following:

- Text entry controls
- Specialized text entry controls
- Submit and reset buttons

- Radio and checkbox groups
- Pull-down and scrolling menus
- File selection and upload controls
- Hidden controls
- Date and time controls
- Numerical controls
- Color picker controls

We'll pause along the way to allow you to try them out by constructing the pizza ordering form shown in FIGURE 9-2.

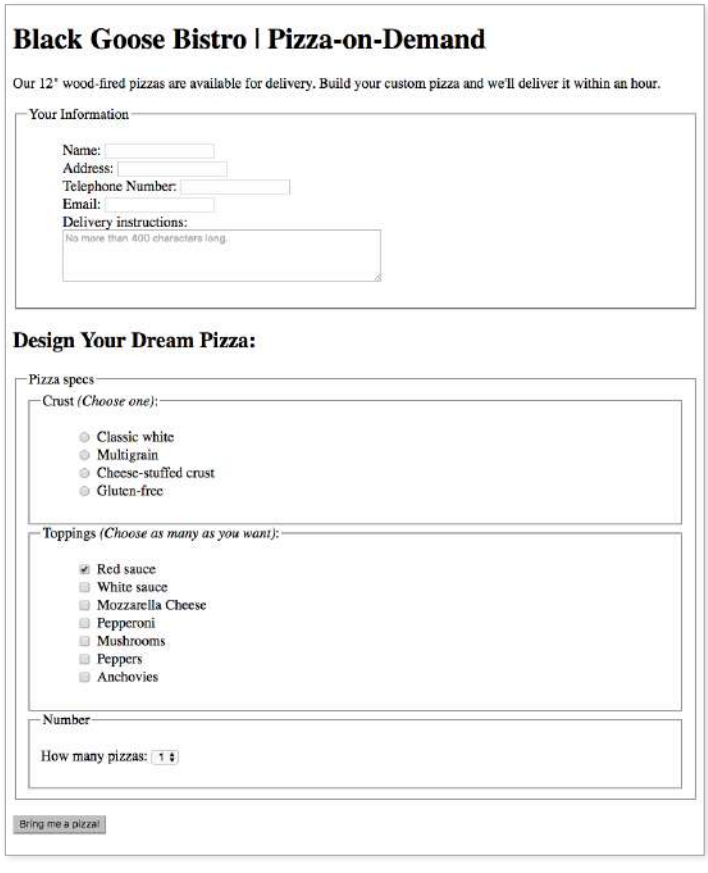


FIGURE 9-2. The pizza ordering form we'll build in the exercises in this chapter

NOTE

The attributes associated with each input type are listed in TABLE 9-1 at the end of this chapter.

As you will see, the majority of controls are added to a form via the **`input`** element. The functionality and appearance of the **`input`** element changes based on the value of its **`type`** attribute. In the current HTML specification, there are 22 types of input controls. We'll take a look at them all (see **Note**).

Text Entry Controls

One of the most common web form tasks is entering text information. Which element you use to collect text input depends on whether users are asked to enter a single line of text (**input**) or multiple lines (**textarea**).

Be aware that if your form has text entry fields, it must use the secure HTTPS protocol to protect user-entered content while the data is in transit to the server (see the **"Moving from HTTP to Secure by Default"** sidebar in **Chapter 2**).

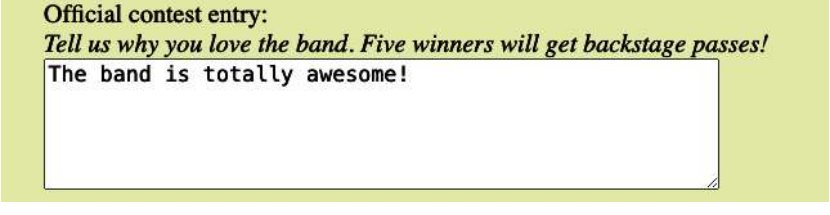
Single-line text field

One of the most straightforward form input types is the text entry field for entering a single word or line of text. In fact, it is the default input type, which means it is what you'll get if you forget to include the **type** attribute or include an unrecognized value. Add a text input field to a form by inserting an **input** element with its **type** attribute set to "text", as shown here and in the top example in FIGURE 9-3 (see **Note**):

```
<input type="text" name="favcolor" value="Red" maxlength="50">
```

Text-entry field (`input type="text"`)

Multiline text-entry field with text content
(`<textarea ...>The band is totally awesome!"</textarea>`)



Multiline text-entry field with placeholder text
(`<textarea ... placeholder="50 words or less" ...></textarea>`)

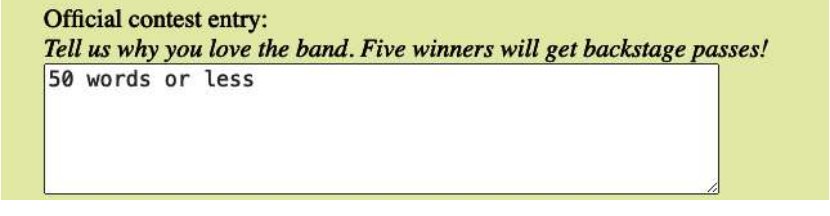


FIGURE 9-3. Examples of the text entry control options for web forms

NOTE

The specific rendering style of form controls varies by operating system and browser version.

NOTE

The markup examples throughout this section include the **label** *element, which is used to improve accessibility. I'll discuss* **label** *in the upcoming* **"Form Accessibility Features"** *section, but in the meantime, I want you to get used to seeing proper form markup.*

`<input type="text">`
Single-line text entry control

There are a few attributes in the **input** element that I'd like to point out:

name

The **name** attribute is required for indicating the variable name.

value

The **value** attribute specifies default text that appears in the field when the form is loaded. When you reset a form, it returns to this value. The value of the **value** attribute gets submitted to the server, so in this example, the value "Red" will be sent with the form unless the user changes it. As an alternative, you could use the **placeholder** attribute to provide a hint of what to type in the field, such as "My favorite color." The value of **placeholder** is not submitted with the form; this is purely a user interface enhancement. You'll see it in action in the upcoming section.

maxlength, minlength

By default, users can type an unlimited number of characters in a text field, regardless of its size (the display scrolls to the right if the text exceeds the character width of the box). You can set a maximum character limit using the **maxlength** attribute if the form processing program you are using requires it. The **minlength** attribute specifies the minimum number of characters.

size

The **size** attribute specifies the length of the rendered input field measured in number of visible characters. It is more common, however, to use style sheets to set the size of the input area. By default, a text input widget typically displays at a size that accommodates 20 characters.

Multiline text entry field

`<textarea>...</textarea>`
Multiline text entry control

At times, you'll want your users to be able to enter more than just one line of text. For these instances, use the **textarea** element, which is replaced by a multiline, scrollable text entry box in the browser (FIGURE 9-3).

Unlike the **input** element (which is void), you can put content between the opening and closing tags in the **textarea** element:

```
<textarea name="contest_entry" rows="5" cols="50">The band is totally
awesome!</textarea>
```

The content of the **textarea** element shows up in the text box when the form is displayed in the browser. It also gets sent to the server when the form is submitted, so carefully consider what goes there.

The **rows** and **cols** attributes provide a way to specify the size of the **textarea** with markup. **rows** specifies the number of lines the text area should display, and **cols** specifies the width in number of characters, although it is more common to use CSS to specify the width of the field. Scrollbars will display if the user types more text than fits in the allotted space.

There are also a few attributes not shown in the example. The **wrap** attribute specifies whether the soft line breaks (where the text naturally wraps at the edge of the box) are preserved when the form is submitted. A value of **soft** (the default) does not preserve line breaks. The **hard** value preserves line breaks when the **cols** attribute is used to set the character width of the box. The **maxlength** and **minlength** attributes set the maximum and minimum number of characters that can be typed into the field.

It is not uncommon for developers to put nothing between the opening and closing tags, and provide a hint of what should go there with a **placeholder** attribute instead:

```
<textarea name="contest_entry" placeholder="50 words or less" rows="5"
cols="50"></textarea>
```

Placeholder text, unlike **textarea** content, is not sent to the server when the form is submitted. Examples of **textarea** content and placeholder text are shown in FIGURE 9-3.

Specialized Text Entry Fields

In addition to the generic single-line text entry field, there are a number of input types for entering specific types of information such as passwords, search terms, email addresses, telephone numbers, and URLs.

Password entry field

```
<input type="password">
```

Password text control

A password field works just like a text entry field, except the characters are obscured from view by asterisk (*) or bullet (•) characters, or another character determined by the browser.

It's important to note that although the characters entered in the password field are not visible to casual onlookers, the form does not encrypt the information, so this should not be considered a real security measure.

disabled and readonly

The **disabled** and **readonly** attributes both prevent users from interacting with a form control, but they work slightly differently.

When a form element is disabled, it cannot be selected. Visual browsers may render the control as grayed-out by default (which you can change with CSS). The disabled state can only be changed with a script. This is a useful attribute for restricting access to some form fields based on data entered earlier in the form and can be applied to any form control or **fieldset**.

The **readonly** attribute prevents the user from changing the value of the form control (although it can be selected). This enables developers to use scripts to set values for controls contingent on other data entered earlier in the form. Inputs that are **readonly** should have strong visual cues that they are somehow different from other inputs, or they could be confusing to users who are trying to change their values. The **readonly** attribute can be used with **textarea** and text-based input controls (see TABLE 9-1 at the end of this chapter).

The important difference is that **readonly** fields are submitted when the form is submitted, but **disabled** ones are not.

Here is an example of the markup for a password field. FIGURE 9-4 shows how it might look after the user enters a password in the field:

```
<input type="password" maxlength="12" name="password" id="form-pswd">
```

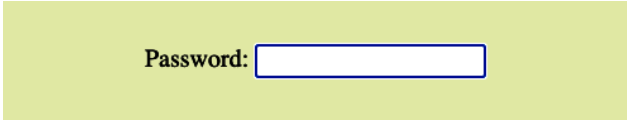


FIGURE 9-4. Passwords are converted to bullets in the browser display.

Search, email, telephone number, and URL fields

`<input type="search">`
Search

`<input type="email">`
Email address

`<input type="tel">`
Telephone number

`<input type="url">`
Location (URL)

The **search**, **email**, **tel**, and **url** input types give the browser a heads-up as to what type of text information to expect in the field. These input types use the same attributes as the generic text input type described earlier (**name**, **maxlength**, **minlength**, **size**, and **value**), as well as a number of other attributes (see TABLE 9-1 at the end of the chapter).

All of these input types typically display as single-line text inputs, but browsers can do some interesting things with the extra semantic information. For example, mobile browsers use the input type to provide a keyboard well suited to the entry task, such as the keyboard featuring a search button for the **search** input type or a ".com" button when the input type is set to **url** (FIGURE 9-5).

There may be inconsistencies in the way these widgets are handled. Very old browsers that do not recognize them at all will display the default generic text input instead, which is a perfectly fine fallback.

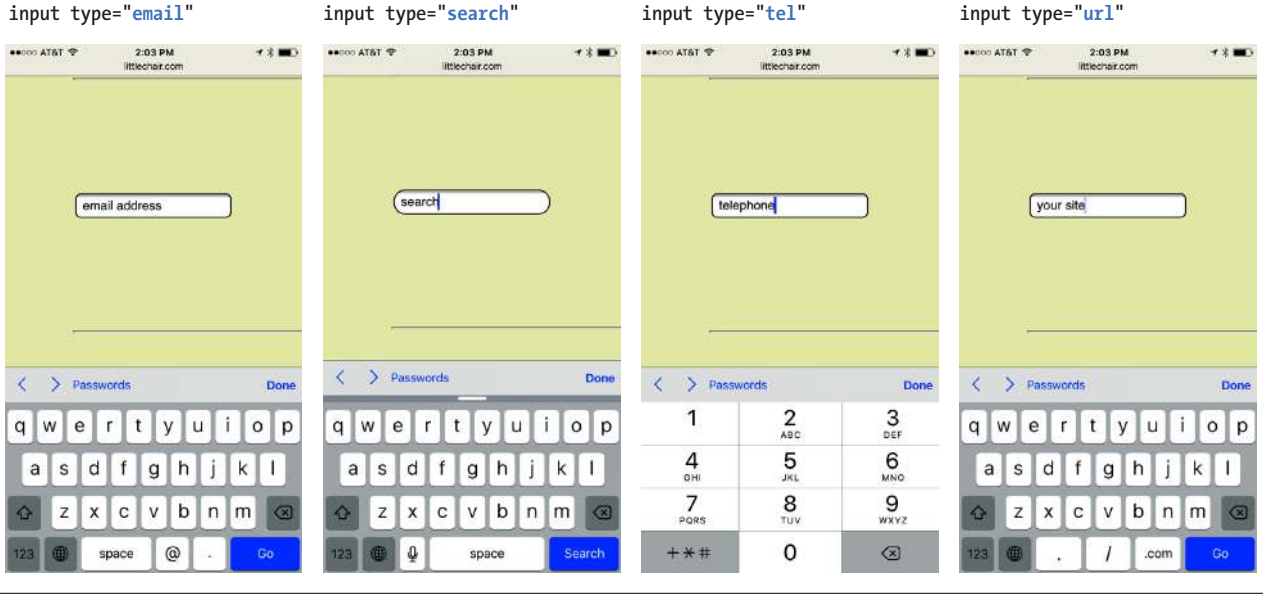


FIGURE 9-5. Mobile browsers, such as Safari on iOS, provide custom keyboards based on the input type.

Autocomplete Suggestions (the datalist element)

Providing suggested options may improve the efficiency and usability of certain text fields. If you want people to be able to type in their own responses but also help them out with a few shortcuts, use a data list. The **datalist** element defines a list of options that pop up when the user puts the cursor in a text field (FIGURE 9-6).

<datalist>...</datalist>

Drop-down menu input

Within the **datalist** element, suggested values are marked up as **option** elements. Use the **list** attribute in the **input** element to associate it with the **id** of its respective **datalist**.

In the following example (FIGURE 9-6), a **datalist** suggests several education level options for a text input:

```
<p>Education completed: <input type="text" list="edulevel"
name="education"></p>

<datalist id="edulevel">
  <option value="High School">
  <option value="Bachelors Degree">
  <option value="Masters Degree">
  <option value="PhD">
</datalist>
```

Be forewarned that **datalist** is not supported by all major browsers as of this writing. Search *CanIUse.com* for up-to-date support information.

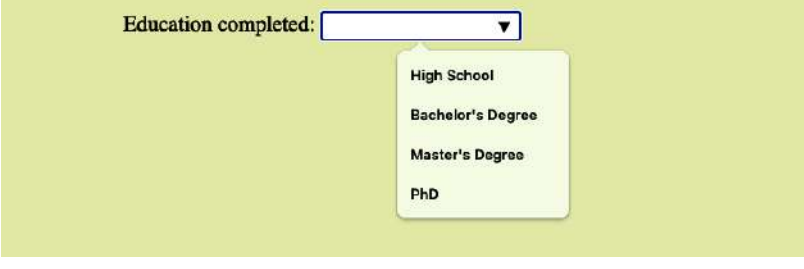


FIGURE 9-6. A **datalist** creates a pop-up menu of suggested values for a text entry field.

WARNING

The values from form controls should be checked by the server code (PHP, Node.js, etc.), as they can be hacked or manipulated. So, although the specialized input types make controlling and validating user input easier, it is still vital to perform server-side checks before updating the database on the server.

Submit and Reset Buttons

`<input type="submit">`
Submits the form data to the server

`<input type="reset">`
Resets the form controls to their default settings

There are several kinds of buttons that can be added to web forms. The most fundamental is the **submit button**. When clicked or tapped, the submit button immediately sends the collected form data to the server for processing. A **reset button** returns the form controls to the state they were in when the form initially loaded. In other words, resetting the form doesn't simply clear all the fields. These buttons have irreversible impacts when they are activated—everything the user has entered so far is sent off or wiped out entirely—so be sure they are used deliberately and not placed where they may be clicked accidentally.

Both submit and reset buttons are added via the `input` element. As mentioned earlier, because these buttons have specific functions that do not include the entry of data, they are the only form control elements that do not require the `name` attribute, although it's OK to add one if you need it.

Submit and reset buttons are straightforward to use. Just place them in the appropriate place in the form, which in most cases is at the very end. By default, the submit button displays with the label "Submit" or "Submit Query," and the reset button is labeled "Reset." You can change the text on the button by using the `value` attribute, as shown for the reset button in this example (FIGURE 9-7).

```
<p><input type="submit"> <input type="reset" value="Start over"></p>
```

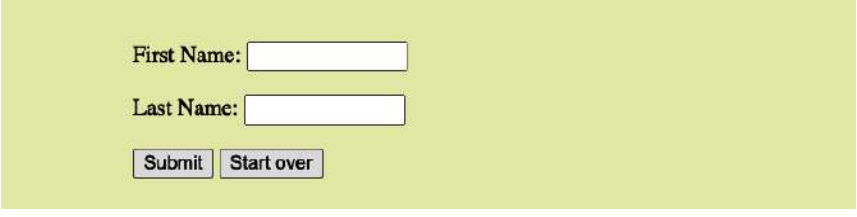


FIGURE 9-7. Submit and reset buttons

The reset button is no longer used in forms as commonly as it used to be, and it should be used with caution. This is because in contemporary form development, we use JavaScript to check the validity of form inputs along the way, so users get feedback as they go along. With thoughtful design and assistance, fewer users should get to the end of the form and need to reset the whole thing. Still, it is a function to be aware of.

At this point, you know enough about form markup to start building the questionnaire shown in FIGURE 9-2 at the beginning of the chapter.

EXERCISE 9-1 walks you through the first steps.

A Few More Buttons

A handful of custom buttons are a bit more advanced to use, but here they are for the sake of thoroughness:

Image buttons
`<input type="image">`

This control allows you to replace the submit button with an image of your choice. The image will appear flat, not like a 3D button. This type of button has accessibility issues, so be sure to include an `alt` value.

Custom input button
`<input type="button">`

Setting the type of the `input` element to "button" creates a button that can be customized with JavaScript. It has no predefined function on its own.

The button element
`<button>...</button>`

The `button` element is a flexible element for creating custom buttons. The content of the `button` element (text and/or images) is what gets displayed on the button.

Because of their flexibility, there's a lot you can do with a simple `button` element. For more information on this, and when to use one instead of a link, read "A Complete Guide to Links and Buttons" by Chris Coyier (*css-tricks.com/a-complete-guide-to-links-and-buttons/*) and "Links vs. Buttons in Modern Web Applications" by Marcy Sutton (*marcysutton.com/links-vs-buttons-in-modern-web-applications*).

EXERCISE 9-1. Starting the pizza ordering form

Here's the scenario. You are the web developer in charge of creating an online pizza ordering form for Black Goose Bistro. The owner has handed you a sketch (FIGURE 9-8) of the form's content. There are sticky notes from the server administrator with information about the script and variable names you need to use with the server application.

Your challenge is to turn the sketch into a functional form. I've given you a head start by creating a bare-bones document with text content and minimal markup and styles. This document, *pizza.html*, is available online at *learningwebdesign.com/6e/materials*. The finished form is also provided.

1. Open the file *pizza.html* in a text or code editor.

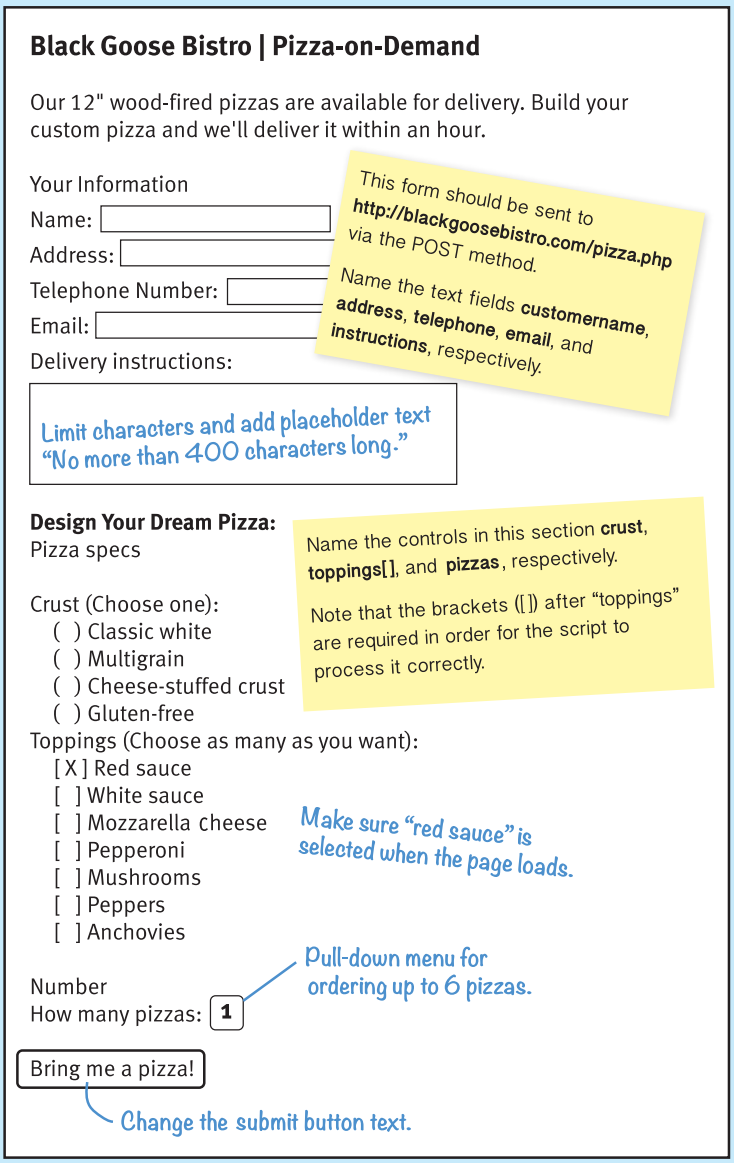

Black Goose Bistro | Pizza-on-Demand
Our 12" wood-fired pizzas are available for delivery. Build your custom pizza and we'll deliver it within an hour.
Your Information
Name:
Address:
Telephone Number:
Email:
Delivery instructions:
This form should be sent to http://blackgoosebistro.com/pizza.php via the POST method.
Name the text fields customername, address, telephone, email, and instructions, respectively.
Limit characters and add placeholder text "No more than 400 characters long."
Design Your Dream Pizza:
Pizza specs
Name the controls in this section crust, toppings[], and pizzas, respectively.
Note that the brackets ([]) after "toppings" are required in order for the script to process it correctly.
Crust (Choose one):
( ) Classic white
( ) Multigrain
( ) Cheese-stuffed crust
( ) Gluten-free
Toppings (Choose as many as you want):
[X] Red sauce
[ ] White sauce
[ ] Mozzarella cheese
[ ] Pepperoni
[ ] Mushrooms
[ ] Peppers
[ ] Anchovies
Make sure "red sauce" is selected when the page loads.
Number
How many pizzas: 1
Pull-down menu for ordering up to 6 pizzas.
Bring me a pizza!
Change the submit button text.

FIGURE 9-8. A sketch of the Black Goose Bistro pizza ordering form

EXERCISE 9-1. Continued

2. The first thing we'll do is put everything after the intro paragraph into a `form` element. The administrator has left a note specifying the `action` and the `method` to use for this form. The resulting `form` element should look like this (keep it on one line):

```
<form action="http://www.blackgoosebistro.com/
pizza.php" method="post">
...
</form>
```

3. In this exercise, we'll work on the "Your Information" section of the form. Start with the first five short text entry form controls that are marked up appropriately as an unordered list. Here's the first one; you insert the other four:

```
<li>Name: <input type="text" name="customername">
</li>
```

   Hint: Choose the most appropriate input type for each entry field. Be sure to name the input elements as specified in the developer's note.

4. After "Delivery instructions:" add a line break and a multiline text area with no content. Because we aren't writing a style sheet for this form, use markup to make it 4 rows long and 60 characters wide (in the real world, CSS is preferable because it gives you more fine-tuned control):

```
<li>Delivery instructions:<br>
<textarea name="instructions" rows="4" cols="60"
maxlength="400" placeholder="No more than 400
characters long"></textarea></li>
```

5. We'll skip the rest of the form until you get a few more controls under your belt, but you can add the submit button at the end, just before the `</form>` tag. Note that the owner has asked us to change the text on the submit button:

```
<p><input type="submit" value="Bring me a
pizza!"></p>
```

6. Now, save the document and open it in a browser. The parts that are finished should generally match FIGURE 9-2. If they don't, then you have some more work to do.

Once the document looks right, take it for a spin by entering some information and submitting the form. You should get a response like the one shown in FIGURE 9-9. Yes, *pizza.php* actually works, but sorry, no pizzas will be delivered.

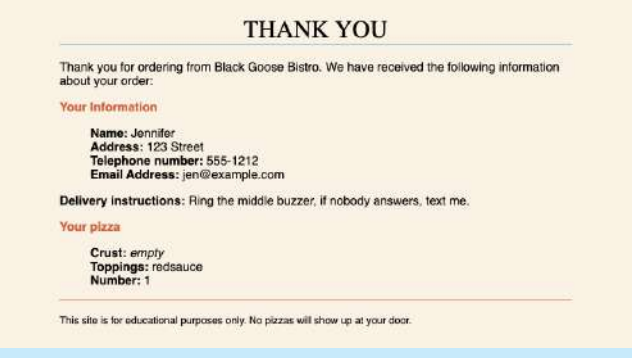

THANK YOU
Thank you for ordering from Black Goose Bistro. We have received the following information about your order:
Your Information
Name: Jennifer
Address: 123 Street
Telephone number: 555-1212
Email Address: jen@example.com
Delivery instructions: Ring the middle buzzer, if nobody answers, text me.
Your pizza
Crust: empty
Toppings: redsauce
Number: 1
This site is for educational purposes only. No pizzas will show up at your door.

FIGURE 9-9. You should see a response page similar to this if your form is working. The pizza description fields will be added in later exercises, so they are set to "empty" or the default values for now.

Radio and Checkbox Groups

Checkboxes and radio buttons make it simple for your visitors to choose from a number of provided options. They are similar in that they function like little on/off switches that can be toggled by the user and are added with the `input` element. They serve distinct functions, however.

Radio groups are appropriate when only one option from the group is permitted.

A form control made up of a collection of **radio buttons** is appropriate when only one option from the group is permitted—in other words, when the selections are mutually exclusive (such as Yes/No, or Pick-up/Delivery). When one radio button is "on," all of the others must be "off," sort of the way buttons used to work on old radios: press one button in, and the rest pop out.

When checkboxes are grouped together, however, it is possible to select as many or as few from the group as desired. This makes them the right choice for lists in which more than one selection is OK.

Radio group

<input type="radio">
Radio button

Radio buttons are added to a form via the **input** element with the **type** attribute set to "radio". Here is the syntax for a minimal radio button:

```
<input type="radio" name="variable" value="value">
```

The **name** attribute is required and plays an important role in binding multiple radio inputs into a set. When you give a number of radio button inputs the same **name** value, they create a group of mutually exclusive options.

In this example (see **Note**), radio buttons are used as an interface for users to enter their age group. A person can't belong to more than one age group, so radio buttons are the right choice:

```
<p>How old are you?</p>
<ol>
  <li><input type="radio" name="age" value="under24" checked> under
24</li>
  <li><input type="radio" name="age" value="25-34"> 25 to 34</li>
  <li><input type="radio" name="age" value="35-44"> 35 to 44</li>
  <li><input type="radio" name="age" value="over45"> 45+</li>
</ol>
```

Notice that although all of the **input** elements have the same variable name ("age"), their values are different. Because these are radio buttons, only one button can be checked at a time, and therefore, only one value will be sent to the server for processing when the form is submitted.

You can decide which button is checked when the form loads by adding the **checked** attribute to the **input** element (see **Note**). In this example, the button next to "under 24" will be checked when the page loads. FIGURE 9-10 shows how radio buttons are rendered in the browser.

Radio buttons (`input type="radio"`) Checkboxes (`input type="checkbox"`)

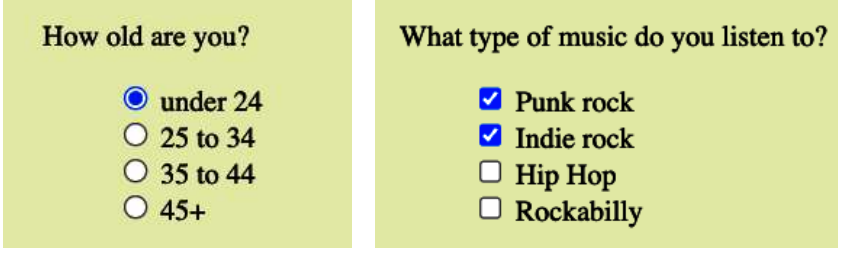


FIGURE 9-10. Radio buttons (left) are appropriate when only one selection is permitted. Checkboxes (right) are best when users may choose any number of options, from none to all of them.

NOTE

I have chosen to omit the markup required for form accessibility (`label` *and* `fieldset`*) from the examples in this section in order to keep them as simple and clear as possible. In the real world, all inputs should have labels. The upcoming section* **"Form Accessibility Features"** *provides details on these important features.*

NOTE

It may look like the **checked** *attribute has no value, but it is one of the attributes in HTML that can be* **minimized** *to one word. Behind the scenes, the minimized* **checked** *attribute stands for the rather redundant:*

```
checked="checked"
```

Checkboxes

`<input type="checkbox">`
Checkbox button

Checkbox groups allow more than one item to be selected.

Checkboxes are added via the **input** element with its type set to **checkbox**. As with radio buttons, you create groups of checkboxes by assigning them the same **name** value. The difference, as I've already noted, is that more than one checkbox may be checked at a time. The value of every checked checkbox will be sent to the server when the form is submitted. Here's an example of a group of checkbox buttons used to indicate musical interests:

```
<p>What type of music do you listen to?</p>
<ul>
  <li><input type="checkbox" name="genre" value="punk" checked> Punk
rock</li>
  <li><input type="checkbox" name="genre" value="indie" checked> Indie
rock</li>
  <li><input type="checkbox" name="genre" value="hiphop"> Hip Hop</li>
  <li><input type="checkbox" name="genre" value="rockabilly">
Rockabilly</li>
</ul>
```

Checkbox buttons may also use the **checked** attribute to make them preselected when the form loads; FIGURE 9-10 shows how they look in the browser.

Checkboxes don't need to be used in groups, of course. In this example, a single checkbox is used to allow visitors to opt in to special promotions. The value of the control will be passed along to the server only if the user checks the box:

```
<p><input type="checkbox" name="optIn" value="yes"> Yes, send me news
and special promotions by email.</p>
```

In EXERCISE 9-2, you'll get a chance to add both radio and checkbox buttons to the pizza ordering form.

EXERCISE 9-2. Adding radio buttons and checkboxes

The next section of the Black Goose Bistro pizza ordering form uses radio and checkbox groups for selecting pizza options. Open the *pizza.html* document and follow these steps, referring to the instructions and content in FIGURE 9-8:

1. In the "Design Your Dream Pizza" section, there are lists of Crust and Toppings options. The Crust options should be radio buttons, because pizzas have only one crust. Insert a radio button before each option. Follow this example for the remaining Crust options:

   ```
   <li><input type="radio" name="crust" value="white"> Classic white</li>
   ```

2. Mark up the Toppings options as you did the Crust options, but this time, the **type** should be **checkbox**. Make sure that the variable name for each is **toppings[]** (see **Note**) and that the "Red sauce" option is preselected (**checked**), as noted on the sketch in EXERCISE 9-1.

3. Save the document and check your work by opening it in a browser to make sure it looks right; then submit the form to make sure it's functioning properly.

NOTE

The **toppings[]** *variable name has brackets at the end because it follows a PHP naming convention. It is only a coincidence that this resembles a checkbox.*

Menus

Another way to provide a list of choices is to put them in a drop-down or scrolling menu. Menus tend to be more compact than radio and checkbox groups.

You add both drop-down and scrolling menus to a form with the **select** element. Whether the menu pulls down or scrolls is the result of how you specify its size and whether you allow more than one option to be selected. Let's take a look at both menu types.

`<select>...</select>`
Menu control

`<option>...</option>`
An option within a menu

`<optgroup>...</optgroup>`
A logical grouping of options within a menu

Drop-down menus

The **select** element displays as a **drop-down menu** (also called a **pull-down menu**) by default when no size is specified or if the **size** attribute is set to 1. In pull-down menus, only one item may be selected. Here's an example (shown in FIGURE 9-11):

```
<p>What is your favorite 80s band?
<select name="EightiesFave">
   <option>The Cure</option>
   <option>Cocteau Twins</option>
   <option>Tears for Fears</option>
   <option>Thompson Twins</option>
   <option value="EBTG">Everything But the Girl</option>
   <option>Depeche Mode</option>
   <option>The Smiths</option>
   <option>New Order</option>
</select>
</p>
```

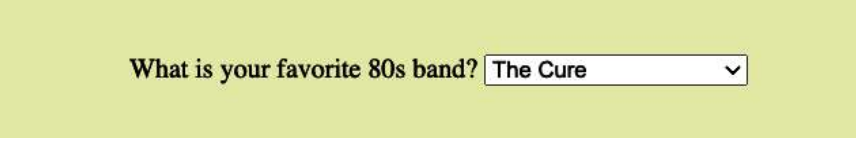


FIGURE 9-11. Pull-down menus pop open when the user clicks the arrow or bar.

You can see that the **select** element is just a container for a number of **option** elements. The content of the chosen **option** element is what gets passed to the web application when the form is submitted. If, for some reason, you want to send a different value than what appears in the menu, use the **value** attribute to provide an overriding value. For example, the preceding code specifies that if someone selects "Everything But the Girl" from the sample menu, the form submits the value "EBTG" for the "EightiesFave" variable. For the other choices, the content between the **<option>** tags will be sent as the value.

Scrolling menus

To make the menu display as a scrolling list, simply specify the number of lines you'd like to be visible using the **size** attribute. This example menu has

the same options as the previous one, except it has been set to display as a scrolling list that is six lines tall (FIGURE 9-12):

```
<p>What 80s bands did you listen to?
<select name="EightiesBands" size="6" multiple>
   <option>The Cure</option>
   <option>Cocteau Twins</option>
   <option selected>Tears for Fears</option>
   <option selected>Thompson Twins</option>
   <option value="EBTG">Everything But the Girl</option>
   <option>Depeche Mode</option>
   <option>The Smiths</option>
   <option>New Order</option>
</select>
</p>
```

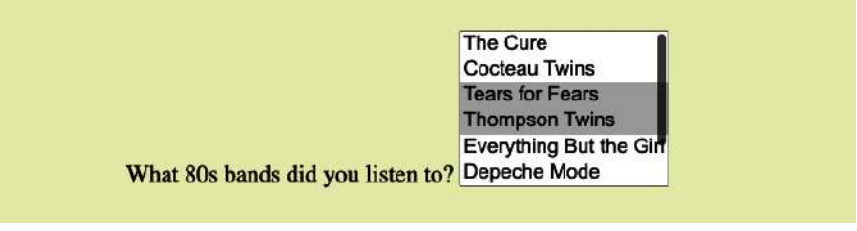


FIGURE 9-12. A scrolling menu with multiple options selected

You may notice a few minimized attributes tucked in there. The **multiple** attribute allows users to make more than one selection from the scrolling list. Note that pull-down menus do not allow multiple selections; when the browser detects the **multiple** attribute, it displays a small scrolling menu automatically by default.

Use the **selected** attribute in an **option** element to make it the default value for the menu control. Selected options are highlighted when the form loads. The **selected** attribute can be used with pull-down menus as well.

Grouping menu options

You can use the **optgroup** element to create conceptual groups of options. The required **label** attribute provides the heading for the group (see **Note**). FIGURE 9-13 shows how the option groups defined here are rendered in modern browsers:

```
<select name="icecream" size="7" multiple>
  <optgroup label="traditional">
    <option>vanilla</option>
    <option>chocolate</option>
  </optgroup>
  <optgroup label="fancy">
    <option>Super praline</option>
    <option>Nut surprise</option>
    <option>Candy corn</option>
  </optgroup>
</select>
```

NOTE

The **label** *attribute in the* **optgroup** *element is not the same as the* **label** *element used to improve accessibility (discussed later in this chapter).*

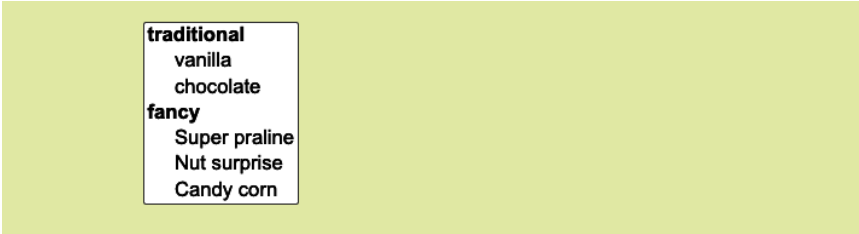


FIGURE 9-13. Option groups

In EXERCISE 9-3, you will use the **select** element to let Black Goose Bistro customers choose the number of pizzas they want to order.

File Selection Control

`<input type="file">`

File selection field

Web forms can collect more than just data. They can also be used to transmit external documents from a user's hard drive. For example, a printing company could use a web form to upload artwork for a business card order, or a magazine could use a form to collect digital photos for a photo contest.

The file selection control makes it possible for users to select a document from the hard drive to be submitted with the form data. We add it to the form with our old friend, the **input** element, with its **type** set to **file**.

The markup sample here (FIGURE 9-14) shows a file selection control used for photo submissions:

```
<form action="/client.php" method="post" enctype="multipart/form-data">
  <label>Send a photo to be used as your online icon <em>(optional)</
em>:
  <input type="file" name="photo"></label>
</form>
```

The appearance of the file upload widget varies slightly by browser and operating system, but it is generally a button that allows you to access the file organization system on your computer (FIGURE 9-14).

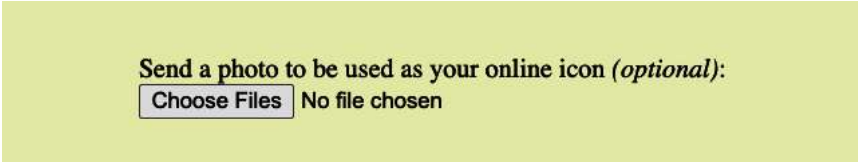


FIGURE 9-14. A file selection form field

EXERCISE 9-3. Adding a menu

The last type of control to add to the pizza ordering form is a pull-down menu for selecting the number of pizzas to have delivered.

1. Insert a **select** menu element with the option to order between 1 and 6 pizzas:

```
<p>How many pizzas:
<select name="number" size="1">
    <option>1</option>
<-- more options here -->
</select>
</p>
```

2. Save the document and check it in a browser. You can submit the form, too, to be sure that it's working. You should get the "Thank You" response page listing all of the information you entered in the form.

Congratulations! You've built your first working web form. Hang on to this document. In EXERCISE 9-4, we'll add markup that makes it more accessible to assistive devices.

It's important to note that when a form contains a file selection input element, you must specify the encoding type (**enctype**) as "multipart/form-data" in the **form** element and use the POST method.

The file input type has a few attributes. The **accept** attribute gives the browser a heads-up on what file types may be accepted (audio, video, image, or some other format identified by its media type). Adding the **multiple** attributes allows multiple files to be selected for upload. The **required** attribute, as its name suggests, requires a file to be selected.

Hidden Controls

`<input type="hidden">`
Hidden control field

There may be times when you need to send information to the form processing application that does not come from the user. In these instances, you can use a hidden form control that sends data when the form is submitted, but is not visible when the form is displayed in a browser (see **Warning**).

A hidden control is added via the **input** element with the **type** set to **hidden**. Its sole purpose is to pass a name/value pair to the server when the form is submitted. In this example, a hidden form element is used to provide the location of the appropriate thank you document to display when the transaction is complete:

```
<input type="hidden" name="success-link" value="http://www.example.com/
thankyou.html">
```

I've worked with forms that have had dozens of hidden controls in the **form** element before getting to the parts that the user actually fills out. This is the kind of information you get from the application developer, system administrator, script documentation, or whoever is helping you get your forms processed. If you are using an existing script, be sure to check the accompanying instructions to see if any hidden form variables are required.

WARNING

It is possible for users to access and manipulate hidden form controls. If you become a professional web developer, you will learn to program "defensively" to avoid this sort of problem. Programming defensively means writing code in a manner that compensates if unexpected things happen or go wrong.

Date and Time Controls

`<input type="date">`
Date input control

`<input type="time">`
Time input control

`<input type="datetime-local">`
Date/time control

`<input type="month">`
Specifies a month in a year

`<input type="week">`
Specifies a particular week in a year

If you've ever booked a hotel or a flight online, you've no doubt used a little calendar widget for choosing the date. Chances are, that little calendar was created with JavaScript; however, the HTML specification does include six input types that make date and time selection widgets part of a browser's standard built-in display capabilities. Date and time pickers are available in all modern browsers, but even in very old non-supporting browsers, the date and time input types display as perfectly usable text entry fields instead. FIGURE 9-15 shows various date and time widgets as rendered in Chrome on macOS.

The date- and time-related input types are as follows:

`<input type="date" name="name" value="2025-01-14">`

Creates a date input control, such as a pop-up calendar, for specifying a date (year, month, day). The initial value must be provided in ISO date format (`YYYY-MM-DD`).

`<input type="time" name="name" value="03:13:00">`

Creates a time input control for specifying a time (hour, minute, seconds, and optionally, fractional sections) with no time zone indicated. The value is provided as `hh:mm:ss:fs`.

`<input type="datetime-local" name="name" value="2011-07-28T03:13:00">`

Creates a combined date/time input control with no time zone information (`YYYY-MM-DDThh:mm:ss`).

`<input type="month" name="name" value="2025-01">`

Creates a date input control that specifies a particular month in a year (`YYYY-MM`).

`<input type="week" name="name" value="2025-W2">`

Creates a date input control for specifying a particular week in a year using an ISO week numbering format (`YYYY-W#`).

NOTE

The `value` *attribute is optional but may be included to provide a starting date or time in the widget. It is included here to demonstrate date and time formats.*

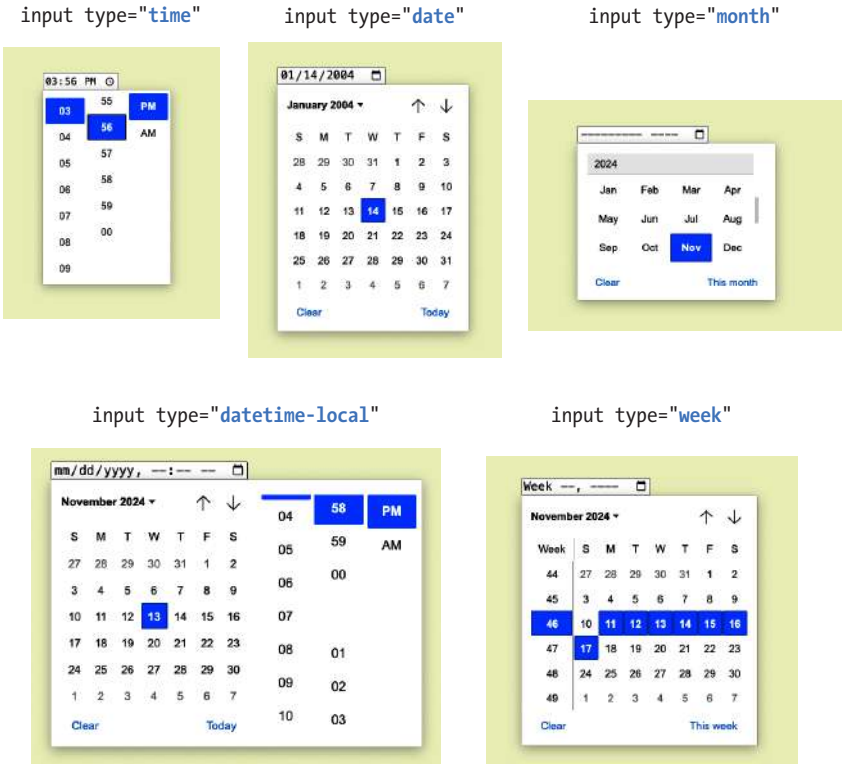


FIGURE 9-15. Date and time picker inputs (shown in Chrome on macOS)

Numerical Inputs

`<input type="number">`
Number input

`<input type="range">`
Slider input

The "number" and "range" input types collect numerical data. For the number input, the browser may supply a spinner widget with up and down arrows for selecting a specific numerical value (a text input may display in user agents that don't support the input type). The range input is typically displayed as a slider (FIGURE 9-16) that allows the user to select a value within a specified range:

```
<label>Number of guests <input type="number" name="guests" min="1"
max="6"></label>

<label>Satisfaction (0 to 10) <input type="range" name="satisfaction"
min="0" max="10" step="1"></label>
```

`input type="number"`

`input type="range"`

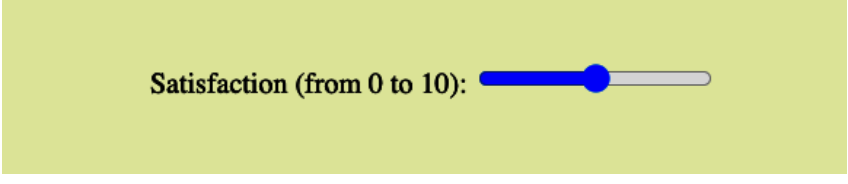


FIGURE 9-16. The number and range input types (shown in Chrome on macOS)

Both the number and range input types accept the **`min`** and **`max`** attributes for specifying the minimum and maximum values allowed for the input (the browser can check that the user input complies with the constraint). Both **`min`** and **`max`** are optional, and you can set one without the other. Negative values are allowed. When the element is selected, the value for both number and range inputs can be increased or decreased with the number keys on a computer keyboard. The range slider can be moved with the mouse or a finger.

The **`step`** attribute allows developers to specify the acceptable increments for numerical input. The default is 1. A value of .5 would permit values 1, 1.5, 2, 2.5, and so on; a value of 100 would permit 100, 200, 300, and so on. You can also set the **`step`** attribute to **`any`** to explicitly accept any value increment.

These two elements allow for only the calculated step values, not for a specified list of allowed values (such as 1, 2, 3, 5, 8, 13, 21). If you need customized values, you'll have to use JavaScript to program that behavior.

Color Selector

<input type="color">
Color picker

The intent of the color input type is to create a pop-up color picker for visually selecting a color value, similar to those used in operating systems or image editing programs. Values are provided as hexadecimal RGB values (**#RRGGBB**). FIGURE 9-17 shows the color picker as rendered in Chrome on macOS (it's the same as the macOS color picker):

```
<label>Your favorite color: <input type="color" name="favorite">
</label>
```

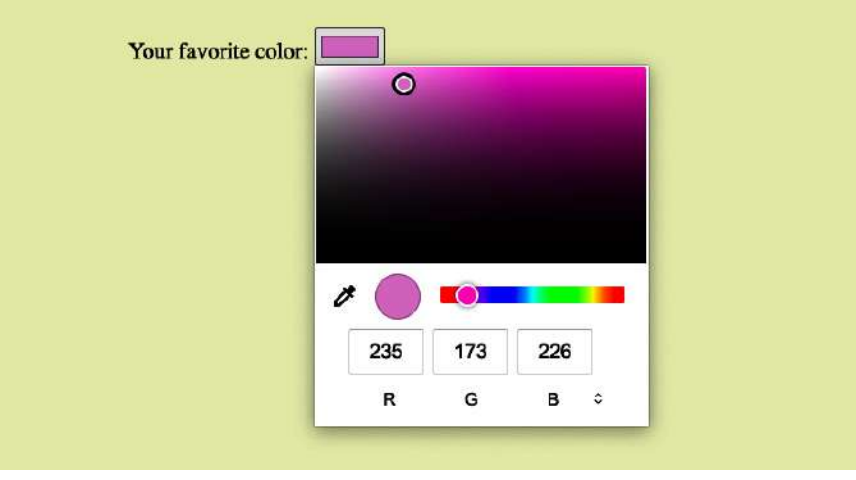


FIGURE 9-17. The color input type (shown in Chrome on macOS)

That wraps up the form control roundup (the sidebar **"A Few More Form Elements"** covers the last few that we haven't discussed). Learning how to insert form controls is one part of the form production process, but any web developer worth their salt will take the time to make sure their forms are as accessible as possible. Fortunately, as you'll see in the following section, there are a few things we can do in markup to describe a form's structure.

FORM ACCESSIBILITY FEATURES

It is essential to consider how users without the benefit of visual browsers will be able to understand and navigate through your web forms. The **label**, **fieldset**, and **legend** form elements improve accessibility by making the semantic connections between the components of a form clear. Not only is the resulting markup more semantically rich, but there are also more elements available to act as "hooks" for style sheet rules. Everybody wins!

A Few More Form Elements

Let's take a quick look at the remaining form elements. You may not be likely to use these right away as a beginner, but I include them here for completeness.

progress

`<progress>...</progress>`
Indicates the state of an ongoing process

The **progress** element gives users feedback on the state of an ongoing process, such as a file download. It may indicate a specific percentage of completion (**determinate**), like a progress bar, or just indicate a "waiting" state (**indeterminate**), like a spinner. The **progress** element requires scripting to function.

```
Percent downloaded: <progress max="100" id="fave">0</progress>
```

meter

`<meter>...</meter>`
Represents a measurement within a range

meter represents a measurement within a known range of values (also known as a **gauge**). It has a number of attributes: **min** and **max** indicate the highest and lowest values for the range (they default to 0 and 100); **low** and **high** could be used to trigger warnings at undesirable levels; and **optimum** specifies a preferred value.

```
<meter min="0" max="100" name="volume">60%</meter>
```

output

`<output>...</output>`
Calculated output value

Simply put, the **output** element indicates the result of a calculation by a script or program. This example, taken from the HTML specification, uses the **output** element and JavaScript to display the sum of numbers entered into inputs a and b.

```
<form onsubmit="return false" oninput="o.value = a.valueAsNumber +
b.valueAsNumber">
  <input name=a type=number step=any>
  + <input name=b type=number step=any> =
  <output name=o for="a b"></output>
</form>
```

Labels

`<label>...</label>`
Attaches information to form controls

Although we may see the label "Address" right next to a text field for entering an address in a visual browser, in the source, the label and field input may be separated. The **label** element associates descriptive text with its respective form field. This provides important context for users with speech-based browsers. Another advantage to using labels is that users can click or tap anywhere on them to select or focus the form control. Users with touch devices will appreciate the larger tap target.

There are two ways to use the **label** element: implicit association and explicit association. In both methods, each **label** element is associated with exactly one form control.

The preferred way to use the **label** element is by using **explicit association**, which matches the label to the control by explicitly referencing its **id**. The **for** attribute in the **label** element says which control the label is for:

```
<label for="form-login-username">Login account</label>
<input type="text" name="login" id="form-login-username">

<label for="form-login-password">Password</label>
<input type="password" name="password" id="form-login-password">
```

This approach is useful when the control is not directly next to its descriptive text in the source. It also offers the potential advantage of keeping the label and the control as two distinct elements, which you may find handy when aligning them with style sheets. The HTML specification states you should use this method whenever possible.

The other method, called **implicit association**, nests the control and its description within a **label** element. The relationship between the label and the control is implied. In the following example, labels are assigned to individual checkboxes and their related text descriptions:

```
<ul>
  <li><label><input type="checkbox" name="genre" value="punk"> Punk
rock</label></li>
  <li><label><input type="checkbox" name="genre" value="indie"> Indie
rock</label></li>
  <li><label><input type="checkbox" name="genre" value="hiphop"> Hip
Hop</label></li>
  <li><label><input type="checkbox" name="genre" value="rockability">
Rockabilly</label></li>
</ul>
```

MARKUP TIP

To keep form-related **id**s distinct from other **id**s on the page, consider prefacing them with "form-", as shown in the examples.

Another technique for keeping forms organized is to give the **form** element itself an **id** and include it as a prefix in the **id**s for the controls it contains, as follows:

```
<form id="form-login">
<input id="form-login-user">
<input id="form-login-pswd">
```

NOTE

You must label individual input elements for checkboxes and radio buttons. There is no way to assign a label to the entire group.

fieldset and legend

The **fieldset** element indicates a logical group of form controls. A fieldset may also include a **legend** element that provides a caption for the enclosed fields.

FIGURE 9-18 shows the default rendering of the following example, but you could use style sheets to change the way the fieldset and legend appear (see **Warning**):

```
<fieldset>
  <legend>Mailing List Sign-up</legend>
  <ul>
    <li><label>Add me to your mailing list <input type="radio"
    name="list" value="yes" checked></label></li>
    <li><label>No thanks <input type="radio" name="list" value="no">
  </label></li>
  </ul>
</fieldset>

<fieldset>
  <legend>Customer Information</legend>
  <ul>
    <li><label>Full name: <input type="text" name="fullname"></label>
    </li>
```

`<fieldset>...</fieldset>`
Groups related controls and labels

`<legend>...</legend>`
Assigns a caption to a fieldset

WARNING

Fieldsets and legends may throw some curveballs when it comes to styling. Be sure to do lots of browser testing if you style these form elements. If you see problems in your target browsers, try putting a **span** *element in the* **legend** *element that you can style without sacrificing accessibility.*

```
    <li><label>Email: <input type="text" name="email"></label></li>
    <li><label>State: <input type="text" name="state"></label></li>
  </ul>
</fieldset>
```

FIGURE 9-18. The default rendering of fieldsets and legends

In EXERCISE 9-4, we'll wrap up the pizza ordering form by making it more accessible with labels and fieldsets.

DIY Form Widgets

Despite having dozens of form widgets straight out of HTML to choose from, it is common for developers to create custom form widgets using markup, CSS, and JavaScript. This might be preferable if you want to provide custom functionality. For example, you could create a drop-down menu using an unordered list inside a **`div`** instead of the standard **`select`** element:

```
<div class="select" role="listbox">
  <ul>
    <li role="option">Red</li>
    <li role="option">Yellow</li>
  </ul>
</div>
```

To help assistive technologies like screen readers recognize this as a form element, the ARIA **`role`** attribute describes the intended function of the **`div`** (a **`listbox`**) and each **`li`** (an **`option`** in that **`listbox`**). There are also many ARIA states and properties that make forms, both standard and custom, usable with assistive devices. For a complete list, see *w3.org/TR/wai-aria/#state_prop_def*.

Custom form widgets require scripting and CSS well beyond the scope of this book, but I wanted you to be aware of the technique. It's also extremely easy to mess up, so create custom controls with caution.

The article "How to Build Custom Form Widgets" on MDN Web Docs (*developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms/How_to_build_custom_form_widgets*) provides a nice overview.

EXERCISE 9-4. Labels and fieldsets

Our pizza ordering form is working, but we need to add labels and fieldsets to make it more usable on assistive devices. Once again, open the *pizza.html* document.

I'll start with the broad strokes and fill in the details later, organizing the form controls into fieldsets and then doing all the labeling. You can do it the other way around if you prefer, or just mark up the labels and fieldsets together as you go.

1. The items in the "Your Information" section at the top of the form are definitely conceptually related, so let's wrap them in a **fieldset** element. Change the markup of the section title from a paragraph (**p**) to a **legend** for the fieldset:

```
<fieldset>
  <legend>Your Information</legend>
  <ul>
    <li>Name: <input type="text" name="fullname"></li>
     ...
  </ul>
</fieldset>
```

2. Next, group the Crust, Toppings, and Number options into one fieldset with the legend "Pizza specs" (again, the text is there; you just need to change it from a **p** to a **legend**):

```
<h2>Design Your Dream Pizza:</h2>
<fieldset>
  <legend>Pizza specs</legend>
  Crust...
  Toppings...
  Number...
</fieldset>
```

3. Create another fieldset just for the Crust options, again changing the description in the **p** element to a **legend**:

```
<fieldset>
  <legend>Crust <em>(Choose one)</em>:</legend>
  <ul>...</ul>
</fieldset>
```

   Do the same for the Toppings and Number sections. In the end, you will have three fieldsets contained within the larger "Pizza specs" fieldset. When you're done, save your document and open it in a browser. It should look very similar to the final form shown back in FIGURE 9-2 (though there may be some minor differences across browsers).

4. OK, now let's get some labels in there. In the "Your Information" fieldset, use explicitly associated labels (**for**/**id**) to tie the text descriptions to each text input. Wrap the description in **label** tags and add the **id** to the **input**. The **for**/**id** values should be descriptive, and they must match. I've done the first one for you; you do the other four:

```
<li><label for="form-name">Name:</label> <input type="text"
name="customername" id="form-name"></li>
```

5. For the radio and checkbox buttons, use implicit association and wrap the **label** element around the **input** and its description. In this way, the button will be selected when the user clicks or taps anywhere inside the **label** element. Here's the first one; you do the rest:

```
<li><label><input type="radio" name="crust" value="white"> Classic
White</label></li>
```

Save your document, and you're done! Labels don't have any effect on how the form looks by default, but you can feel good about the added semantic value you've added and maybe even use them to apply styles at another time.

FORM LAYOUT AND DESIGN

I can't close this chapter without saying a few words about form design, even though this chapter is about markup, not presentation.

Usable Forms

A poorly designed form can ruin a user's experience on your site and negatively impact your business goals. Badly designed forms mean lost customers, so it's critical to get it right—both on the desktop and for small-screen devices with their special requirements. You want the path to a purchase or other action to be as frictionless as possible.

The topic of good web form design is a rich one that could fill a book in itself. In fact, there are several such books! *Web Form Design* (Rosenfeld Media) by web form expert Luke Wroblewski is admittedly getting a bit older, but it holds up very well and I recommend it highly. Luke's subsequent book, *Mobile First* (which you can read for free at **mobile-first.abookapart.com**), includes tips on how to format forms in a mobile context. *Form Design Patterns* (Smashing Books) by Adam Silver is another excellent resource with specific examples of how to design and code user-friendly forms for a variety of common situations.

Here are a few form design tips to get you started (these are from *Web Form Design*):

Avoid unnecessary questions.

Help your users get through your form as easily as possible by not including questions that are not absolutely necessary to the task at hand. Extra questions, in addition to slowing things down, may make a user wary of your motivations for asking. If you have another way of getting the information (for example, the type of credit card can be determined from the first four numbers of the account), then use alternative means and don't put the burden on the user. If there is information that might be nice to have but is not required, consider asking at a later time, after the form has been submitted and you have built a relationship with the user.

Consider the impact of label placement.

The position of the label relative to the input affects the time it takes to fill out the form. The less the user's eye needs to bounce around the page, the quicker the form completion. Putting the labels above their respective fields creates a single alignment for faster scans and completion, particularly when you're asking for familiar information (name, address, etc.). Left alignment of labels results in the slowest form completion, but it may be appropriate if you want the user to slow down or be able to scan and consider the types of required information.

Choose input types carefully.

As you've seen in this chapter, there are quite a few input types to choose from, and sometimes it's not easy to decide which one to use. For example, a list of options could be presented as a `select` (pull-down) menu or a number of choices with checkboxes (see **Note**). Weigh the pros and cons of each control type carefully, and follow up with user testing.

NOTE

*A `select` menu may not be the best tool for the job. Luke Wroblewski shares the results of usability testing in his article "Dropdowns Should Be the UI of Last Resort" (**lukew.com/ff/entry.asp?1950**).*

Group related inputs.

It is easier to parse the many fields, menus, and buttons in a form if they are visually grouped by related topic. For example, a user's contact information could be presented in a compact group so that five or six inputs are perceived as one unit. Usually, all you need is a very subtle indication, such as a fine horizontal rule and some extra space. Don't overdo it.

Clarify primary and secondary actions.

The primary action at the end of the form is usually some form of submit button ("Buy," "Register," etc.) that signals the completion of the form and the readiness to move forward. You want that button to be visually dominant and easy to find (aligning it along the main axis of the form is helpful as well).

Using JavaScript, you can gray out the submit button as nonfunctioning until all necessary data has been filled in.

Secondary actions tend to take you a step back, such as clearing or resetting the form. If you must include a secondary action, make sure that it is styled to look different and less important than the primary action. It is also a good idea to provide an opportunity to undo the action.

Styling Forms

As we've seen in this chapter, the default rendering of form markup is not up to par with the quality we see on most professional web forms today. As for other elements, you can use style sheets to create a clean form layout as well as to change the appearance of most form controls. Something as simple as nice alignment and a look that is consistent with the rest of your site can go a long way toward improving the impression you make on a user.

Keep in mind that form widgets are drawn by the browser and are informed by operating system conventions. However, you can still apply dimensions, margins, fonts, colors, borders, and background effects to form elements such as text inputs, **`select`** menus, text areas, fieldsets, labels, and legends. Be sure to test in a variety of browsers to check for unpleasant surprises. **Chapter 21, More CSS Techniques**, lists some specific techniques that you can try once you have more experience with CSS. For more help, a web search for "CSS for forms" will turn up a number of tutorials.

Dialog Boxes and Disclosure Widgets

As long as we're talking about HTML for adding interactivity to web pages, this might be a good time for a quick introduction to a few more interactive elements. They are not form related, but they are worth knowing about should you find yourself designing or producing a web application.

Dialog boxes

`<dialog>...</dialog>`

Dialog box

The **`dialog`** element is the appropriate markup for a box that pops open with some sort of alert or action item. Typically, dialog boxes should be added to a page with JavaScript, although it is possible to mark one up in the document itself (with limited functionality) using this element. Like other HTML elements, you can style the box and its background with CSS, and you can even animate how it opens and closes.

Disclosure widgets

`<details>...</details>`
`<summary>...</summary>`

Disclosure widget

You've likely seen elements on web pages that you can open and close to read more information. One way to create an element that toggles open and closed is with the **`details`** and **`summary`** elements. The **`summary`** element provides the label for the widget and is what users see when the page loads. Browsers typically display a triangle next to the summary label that indicates it will expand with more content when you click or tap it. The **`details`** element is a container that wraps around the summary and the content that will be revealed when it is opened. Including the **`open`** attribute causes the content of the **`details`** element to be visible when the document loads. You can style the **`details`** and **`summary`** elements with CSS, styling the entire details box and the summary label. By default, the **`summary`** element behaves like a list item when it comes to styling. You can also control the toggle behavior with JavaScript.

```
<details>
  <summary>More info</summary>
  <p>This is some extra information you may be
interested in about disclosure widgets.</p>
</details>
```

Congratulate yourself for making it through this long overview of the form elements! See what you remember in the end-of-chapter quiz.

TEST YOURSELF

Ready to put your web form know-how to the test? Here are a few questions to make sure you've gotten the basics. You'll find the answers in **Appendix A**.

1. Decide whether each of these forms should be sent via the GET or POST method:
   a. A form for accessing your bank account online
   b. A form for sending t-shirt artwork to the printer
   c. A form for searching archived articles
   d. A form for collecting long essay entries
2. Which form control element is best suited for the following tasks? When the answer is **`input`**, be sure to also include the type. Some tasks may have more than one correct answer.
   a. Choose your astrological sign from 12 signs.
   b. Indicate whether you have a history of heart disease (yes or no).
   c. Write up a book review.
   d. Select your favorite ice cream flavors from a list of eight flavors.
   e. Select your favorite ice cream flavors from a list of 25 flavors.
   f. Enter your phone number.
   g. Enter your date of birth for age verification.
3. Each of these markup examples contains an error. Can you spot it?
   a. `<checkbox name="color" value="teal">`
   b.
```
<select name="popsicle">
  <option value="orange">
  <option value="grape">
  <option value="cherry">
</select>
```
   c. `<input type="password">`
   d. `<textarea name="essay" width="100" height="6">Your story.</textarea>`

ELEMENT REVIEW: FORMS

TABLE 9–1 lists all of the form-related elements and attributes included in the current HTML spec (some attributes were not covered in this chapter). The attributes for each input type are listed in TABLE 9–2.

TABLE 9–1. Form-related elements and attributes

Element and attributes	Description
`button`	Generic input button
`autofocus`	Automatically focuses the form control when the page is loaded
`name="text"`	Supplies a unique variable name for the control
`disabled`	Disables the input so it cannot be selected
`type="submit|reset|button"`	Indicates the type of button (the default is **`submit`**)
`value="text"`	Specifies the value to be sent to the server
`menu="id value"`	Specifies a designated pop-up menu
`popovertarget="id of element with popover attribute"`	Targets a popover element to toggle, show, or hide
`popovertargetaction="toggle|show|hide"`	Indicates whether a targeted popover element is to be toggled, shown, or hidden
`form,formaction, formenctype, formmethod, formnovalidate, formtarget`	Form submission-related attributes used for submit and reset buttons
`datalist`	Provides a list of options for text inputs
`fieldset`	Groups related controls and labels
`disabled`	Disables all the inputs in the fieldset so they cannot be selected, edited, or submitted
`form="id value"`	Associates the element with a specific form
`name="text"`	Supplies a unique variable name for the control
`form`	Form element
`action="url"`	Gives the location of the forms processing program (*required*)
`accept-charset="characterset"`	Indicates the character encodings to use
`autocomplete`	Default setting for the autofill feature for controls in the form
`method="get|post|dialog"`	Specifies the method used to submit the form data (the default is **`get`**)
`enctype="content type"`	Specifies the encoding method, generally either **`application/x-www-form-urlencoded`** (the default) or **`multipart/form-data`**
`name="text"`	Provides the name of the form to use in the **`document.forms`** API
`novalidate`	Bypasses form control validation for this form
`target="text|_blank|_self|_parent|_top"`	Sets the browsing context
`rel="HTML link type"`	Controls what kinds of links the elements create (for example, **`noreferrer`** or **`noopener`**)

TABLE 9–1. Continued

Element and attributes	Description
`input`	Creates a variety of controls, based on the **`type`** value
`autofocus`	Indicates the control should be ready for input when the document loads
`type="submit|reset|button|text|password|checkbox|radio|image|file|hidden|email|tel|search|url|date|time|datetime-local|month|week|number|range|color"`	Specifies the type of input (the default is **`text`**)
See **TABLE 9-2** *for a full list of attributes associated with each input type.*	
`disabled`	Disables the input so it cannot be selected, edited, or submitted
`form="form id value"`	Associates the control with a specified form
`label`	Attaches information to controls
`for="text"`	Identifies the associated control by its **`id`** reference
`legend`	Assigns a caption to a **`fieldset`**
`meter`	Represents a fractional value within a known range
`high="number"`	Indicates the range that is considered "high" for the gauge
`low="number"`	Indicates the range that is considered "low" for the gauge
`max="number"`	Specifies the highest value for the range
`min="number"`	Specifies the lowest value for the range
`optimum="number"`	Indicates the number considered to be "optimum"
`value="number"`	Specifies the actual or measured value
`optgroup`	Defines a group of options
`disabled`	Disables the **`optgroup`** so it cannot be selected
`label="text"`	Supplies a label for a group of options
`option`	An option within a **`select`** menu control
`disabled`	Disables the **`option`** so it cannot be selected
`label="text"`	Supplies an alternate label for the option
`selected`	Preselects the option
`value="text"`	Supplies an alternate value for the option
`output`	Represents the results of a calculation
`for="text"`	Creates a relationship between **`output`** and another element
`form="form id value"`	Associates the control with a specified form
`name="text"`	Supplies a unique variable name for the control
`progress`	Represents the completion progress of a task (can be used even if the maximum value of the task is not known)

TABLE 9–1. Continued

Element and attributes	Description
max="*number*"	Specifies the total value or final size of the task
value="*number*"	Specifies how much of the task has been completed
select	Pull-down menu or scrolling list
autocomplete	Provides a hint for the form autofill feature
disabled	Indicates the control is nonfunctional; can be activated with a script
form="*form id value*"	Associates the control with a specified form
multiple	Allows multiple selections in a scrolling list
name="*text*"	Supplies a unique variable name for the control
required	Indicates user input is required for this control
size="*number*"	Specifies the height of the scrolling list, in text lines
textarea	Multiline text entry field
autocomplete	Provides a hint for the form autofill feature
autofocus	Indicates the control should be highlighted and ready for input when the document loads
cols="*number*"	Specifies the width of the text area, in characters
dirname="*text*"	Allows text directionality to be submitted
disabled	Disables the control so it cannot be selected
form="*form id value*"	Associates the control with a specified form
maxlength="*number*"	Specifies the maximum number of characters the user can enter
minlength="*number*"	Specifies the minimum number of characters the user can enter
name="*text*"	Supplies a unique variable name for the control
placeholder="*text*"	Provides a short hint to help the user enter the correct data
readonly	Makes the control unalterable by the user
required	Indicates user input is required for this control
rows="*number*"	Specifies the height of the text area, in text lines
wrap="hard|soft"	Controls whether line breaks in the text input are returned in the data; **hard** preserves line breaks, while **soft** (the default) does not

TABLE 9–2. The attributes available for each `input` element type

Attribute	submit	reset	button	text	password	checkbox	radio	image	file	hidden
accept									•	
alt								•		
autocomplete				•	•					•
autofocus	•	•	•	•	•	•	•	•	•	
checked						•	•			
dirname	•			•	•					•
disabled	•	•	•	•	•	•	•	•	•	•
form	•	•	•	•	•	•	•	•	•	•
formaction	•							•		
formenctype	•							•		
formmethod	•							•		
formnovalidate	•							•		
formtarget	•							•		
height								•		
list				•						
max										
min										
maxlength				•	•				•	
minlength				•	•				•	
multiple									•	
name	•	•	•	•	•	•	•	•	•	•
pattern				•	•					
placeholder				•	•					
readonly				•	•					
required				•	•	•	•		•	
size				•	•				•	
src								•		
step										
value	•	•	•	•	•	•	•		•	•
width								•		

TABLE 9–2. Continued

Attribute	email	telephone, search, url	number	range	date, time, datetime-local, month, week	color
accept						
alt						
autocomplete	•	•	•	•	•	•
autofocus	•	•	•	•	•	•
checked						
dirname	•					
disabled	•	•	•	•	•	•
form	•	•	•	•	•	•
formaction						
formenctype						
formmethod						
formnovalidate						
formtarget						
height						
list	•	•	•	•	•	•
max			•	•	•	
min			•	•	•	
maxlength	•	•				
minlength	•	•				
multiple	•					
name	•	•	•	•	•	•
pattern	•	•				
placeholder	•	•	•			
readonly	•	•	•		•	
required	•	•	•		•	
size	•	•				
src						
step			•	•	•	
value	•	•	•	•	•	•
width						

CHAPTER 10

EMBEDDED MEDIA

IN THIS CHAPTER

The iframe element

The object element

Video and audio players

The canvas element

The HTML specification defines **embedded content** as "content that imports another resource into the document, or content from another vocabulary that is inserted into the document."

In **Chapter 7, Adding Images**, we looked at the `img` and `picture` elements, which "import another resource into the document," and the `svg` element,which inserts an image using "another vocabulary." But images certainly aren't the only things you can stick in a web page. In this chapter, we'll look at other types of embedded content and their respective markup, including the following:

- A window for viewing an external HTML source (`iframe`)
- A multipurpose embedding element (`object`)
- Video and audio players (`video` and `audio`)
- A scriptable drawing area that can be used for animations or game-like interactivity (`canvas`)

WINDOW-IN-A-WINDOW (IFRAME)

`<iframe>...</iframe>`
A nested browsing window

The `iframe` (short for **inline frame**) element lets you embed a separate HTML document or other web resource in a document. It has been around for many years, and it has become one of the most popular ways to share content between sites.

For example, when you request the code to embed a video from YouTube or a map from Google Maps, the sites provide iframe-based code to copy and paste into your page. Many other media sites have followed suit, because it allows them to control aspects of the content you are putting on your page.

Inline frames have also become a standard tool for embedding ads, and web tutorial sites use them to embed code samples in pages.

Adding an **iframe** element to the page creates a little window-in-a-window (or a **nested browsing context**, as it is known in the spec) that displays the external resource. You place an inline frame on a page similarly to an image, specifying the source (**src**) of its content. The **width** and **height** attributes specify the dimensions of the frame. The content in the **iframe** element itself is fallback content for browsers that don't support the element, although virtually all browsers support iframes at this point.

In this very crude example, the parent document displays the web page *glossary.html* in an inline frame (FIGURE 10-1). If **iframe** is not supported, a link to read the glossary is provided. This iframe has its own set of scrollbars because the embedded HTML document is too long to fit in the frame. To be honest, you don't often see iframes used this way in the wild (except for code examples, perhaps), but the example provides a good entry to understanding how they work:

```
<h1>An Inline Frame</h1>

<iframe src="glossary.html" width="400" height="250">
  <a href="glossary.html">Glossary</a>.
</iframe>
```

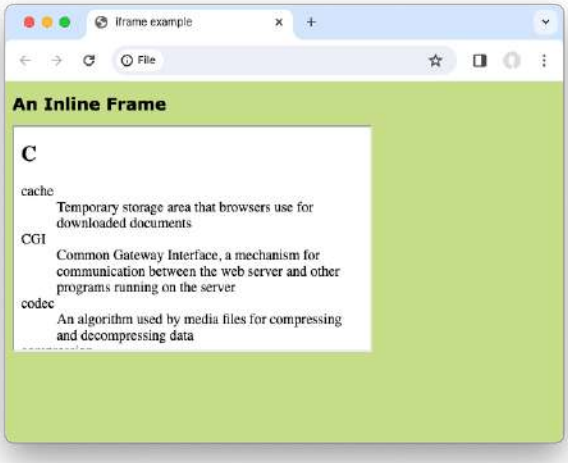

Iframe example
File
An Inline Frame
C
cache
Temporary storage area that browsers use for downloaded documents
CGI
Common Gateway Interface, a mechanism for communication between the web server and other programs running on the server
codec
An algorithm used by media files for compressing and decompressing data

FIGURE 10-1. An inline frame (added with the **iframe** element)

In modern uses of **iframe**, the window is not so obvious. In fact, there is usually no indication that there is an embedded frame there at all, as shown by the Google Maps example in FIGURE 10-2.

FIGURE 10-2. The edges of an iframe are usually not detectable, as shown in this embedded Google Map.

There are some security concerns with using iframes, because they may act like open windows through which hackers can sneak. The **sandbox** attribute puts restrictions on what the framed content can do, such as not allowing forms, pop-ups, scripts, and the like. The **referrer** attribute is another important attribute to help protect your site and users when using iframes; it lets you put limits on how much information the iframe should have access to.

While iframe security is beyond the scope of this chapter, you'll need to brush up if you are going to make use of iframes on your site. I recommend the "Securely Embed Content on Your Site" article on the *Chrome for Developers* blog (***developer.chrome.com/blog/embed-content***) for more information.

To get a feel for how iframes work, use one to embed your favorite video in a page in EXERCISE 10-1.

■ FURTHER READING

To learn more about using iframes, check out the MDN Web Docs article "From object to iframe—Other Embedding Technologies" (***oreil.ly/M3AR1***).

EXERCISE 10-1. Embedding a video with iframe

If you'd like to poke around with an iframe, why not try grabbing a video from YouTube and embedding it in a page?

1. Start by creating a new HTML document, including the basic structural elements that we covered in **Chapter 4, Creating a Simple Page**.
2. Go to YouTube on a desktop browser, and once you are on the page for your chosen video, look for the Share button; then choose the Embed option. The **iframe** code is there for you to copy and paste. If you click "Developer Sample" there will be further configuration options. Just copy the **iframe** code and paste it into the **body** element in the new HTML document. Open it in a browser, and you're done!

Note that this does not work in the YouTube mobile app interface. You may also encounter videos whose creators have not given permission for them to be embedded in your page.

MULTIPURPOSE EMBEDDER (OBJECT)

`<object>...</object>`
Represents an external resource

In the early days, web browsers were extremely limited in what they were able to render, so they relied on **plug-ins** to help them display media that they couldn't handle natively. Java applets, Flash movies, RealMedia (an old web video and audio format) content, and other media required third-party plug-ins in order to be played in the browser. Even displaying JPEG images once required a plug-in!

A plug-in is software that gives a browser functionality that it doesn't have natively.

To embed those media resources in the page, we used the **`object`** and **`embed`** elements. They have slightly different uses. The **`object`** element is a multipurpose object placer. It can be used to place an image, movie, PDF file, or nested browsing context (like an iframe) in the page, or to embed a resource that must be handled by a plug-in. The **`embed`** element was for use with plug-ins only, and it is all but extinct. I won't be covering it in this chapter, but you can find it in the HTML spec. To be honest, there aren't many instances when you'll use **`object`** to embed content these days either, now that we have specialized elements for embedding audio, video, and SVG images. However, some familiarity with **`object`** will serve you well when we use it as one option for adding an SVG to a page in **Chapter 28, SVG**.

NOTE

Earlier versions of the spec included the **`param`** *element for passing information about the media file to the* **`object`** *element, but it has officially been made "obsolete" in favor of the* **`data`** *attribute.*

At its most minimal, the **`object`** element uses the **`data`** attribute (see **Note**) to point to the resource and the **`type`** attribute to provide its MIME type. Any content within the **`<object>`** tags will be used as a fallback for browsers that don't support the embedded resource type. Here is a simple **`object`** element that places a PDF document on the page and provides a link to an online document as a fallback:

```
<object data="veryspecial.pdf" type="application/pdf" width="600"
height="800">
  <p>Read the <a href="special.html">Very Special Document</a>.</p>
</object>
```

Additional attributes for the **`object`** element are available and vary according to the type of media it is placing. See the HTML spec (***html.spec.whatwg.org/iframe-embed-object.html***) for details when you have an occasion to put **`object`** to use.

VIDEO AND AUDIO

In the past, browsers did not have built-in capabilities for handling video or sound, so they used plug-ins to fill in the gap. With the development of the web as an open standards platform, and with broadband connections allowing for heftier downloads than previously, it seemed to be time to make multimedia support part of browsers' out-of-the-box capabilities. Enter the **`video`** and **`audio`** elements and their respective APIs (see the **"API"** sidebar), first introduced in HTML5.

Choosing a Media Format

The good news is that the `video` and `audio` elements have been well supported in all modern browsers for more than 10 years, but there are still some inconsistencies in browser support for various media types. So, before we get to the markup for embedding audio and video, it will be helpful to have a basic understanding of the media file formats that work best for web delivery.

Audio and video files generally have two identifying features. First is the method by which the media content itself has been **encoded** (converted to 1s and 0s) and compressed to a smaller file size. The encoding method is called the **codec**, which is short for "code/decode" or "compress/decompress." There are a bazillion codecs out there (that's an estimate). Some probably sound familiar, like MP3; others might sound new, such as H.264, VP9, and Theora.

Second, there is the **container format** that holds the compressed media and its metadata (information about the codec, bitrate, resolution, and more) together in a package, kind of like a ZIP file. On the web, the most popular container formats for video are MP4 and WebM. For audio, MP3 and WAV should look familiar.

Usually, a container format is compatible with more than one codec type, and when you factor in browser support, the full story gets complicated. The following is a simplified, cut-to-the-chase roundup of media formats best used with the `video` and `audio` embedding elements; however, if you anticipate working with web-delivered and streaming media, you will need to be well versed in the full range of media options available.

Meet the video formats

For video, the most common options are as follows:

MP4 (.mp4 or .m4v)

By far the best-supported media format for web video is MP4 (also referred to as "MPEG-4"), used with the H.264 video codec and the AAC audio codec. The newer H.265 codec (also known as **High Efficiency Video Coding**, or **HEVC**) reduces the bitrate by half, but it's not as well supported as H.264. It should be noted that H.264 is a patented codec, so it must be licensed for commercial uses.

Browser support: Virtually all browsers have supported this format for the last 10 years.

WebM (.webm)

WebM is a container format that has the advantage of being open source and royalty-free. It can be used with the V8, V9, and AV1 video codecs, with V9 being the most popular as of this writing.

Browser support: Chrome, Edge, Firefox, and Opera support WebM fully. Safari was late to the WebM game, offering full support on the desktop in 2022 and only partial support on iOS as of this writing.

■ TERMINOLOGY

API

An **API** (short for **application programming interface**) is a standardized set of commands, data names, properties, actions, and so on that lets one software application communicate with another. HTML5 introduced a number of APIs that give browsers programmable features that previously could only be achieved with third-party plug-ins.

Some APIs have a markup component, such as the `HTMLMediaElement` API for embedding multimedia with the `video` and `audio` elements. Others operate entirely behind the scenes with JavaScript or server-side components, such as the Service Workers API for creating web applications that work even without an internet connection.

The W3C is working on lots and lots of APIs for use with web applications, all in varying stages of completion and implementation. Most have their own specifications, separate from the HTML spec itself. If you're curious, check out this list of APIs maintained on the MDN Web Docs site: *developer.mozilla.org/en-US/docs/Web/API*.

Video and Audio Encoding Tools

There are scores of options for editing and encoding video and audio files, so I can't cover them all here, but the following tools are free and get the job done.

Video conversion

- **HandBrake** (*handbrake.fr*) is a popular open source tool for converting to MPEG4 with H.264, H.265, VP8, and Theora.
- **FFmpeg** (*ffmpeg.org*) is an open source command-line tool for converting just about any video format. If you are not comfortable with the command line, there are a number of software packages that offer a user interface to FFmpeg to make it more user-friendly.
- **Freemake** (*freemake.com*) is a free video and audio conversion tool for Windows that supports over 500 media formats.

Audio conversion

- **Audacity** (*audacityteam.org*) is free, open source, cross-platform audio software for multitrack recording and editing.
- **Audio Converter** (*online-audio-converter.com*) is one of the free audio and video tools from *123Apps.com* that converts files to MP3, WAV, OGG, and more.
- **MediaHuman Audio Converter** (*mediahuman.com/audio-converter/*) is free for macOS and Windows and can convert to all of the audio formats listed in this chapter and more. It has an easy drag-and-drop interface, but is pretty much no-frills.

Meet the audio formats

For audio, the most commonly used formats are:

MP3 (.mp3)

The MP3 (short for MPEG-1 Audio Layer 3) format is a codec and container in one. It has become ubiquitous as a music download format.

Browser support: Virtually all modern browsers support MP3.

WAV (.wav)

The WAV format is also a codec and container in one. This format is uncompressed, so it is best used when you want to provide the highest quality (and largest size) version of an audio file.

Browser support: Virtually all modern browsers support the WAV format.

MPEG-4 (.m4a)

MP4 can be used for audio as well as video and provides higher-quality audio at smaller sizes. Choose the AAC audio codec for best results when using it in the `audio` element.

Browser support: MP4 with AAC is well supported in modern browsers, although it is not as popular as MP3.

WebM (.webm)

The WebM format can contain audio only. Vorbis and Opus are popular codecs for use with WebM.

Browser support: WebM audio support is lacking in Safari as of this writing.

Server Setup

If you are delivering video and audio files, you need to make sure your server software is configured to recognize the MIME type of each media format. In Apache, you add the type to the *.htaccess* file. The following example adds the MP4 type/subtype and extensions:

```
AddType video/mp4 mp4 m4v
```

The MIME types for the media formats covered in this chapter are:

MP4 (H.264)	`video/mp4 mp4 m4v`
WebM (VP9)	`video/webm webm webmv`
MP3	`audio/mpeg mp3`
WAV	`audio/wav audio/wave`
MP4 (AAC)	`audio/mp4 m4a`
WebM (Vorbis or Opus)	`audio/webm webm`

Adding a Video to a Page

`<video>...</video>`
Adds a video player to the page

It's about time we got to the markup for adding a video to a web page (this is an HTML chapter, after all). Let's start with an example that assumes you are designing for an environment where you know exactly what browser your user will be using. When this is the case, you can provide only one video format using the **src** attribute in the **video** tag (just as you do for an **img**). FIGURE 10-3 shows the default movie player in the Chrome browser.

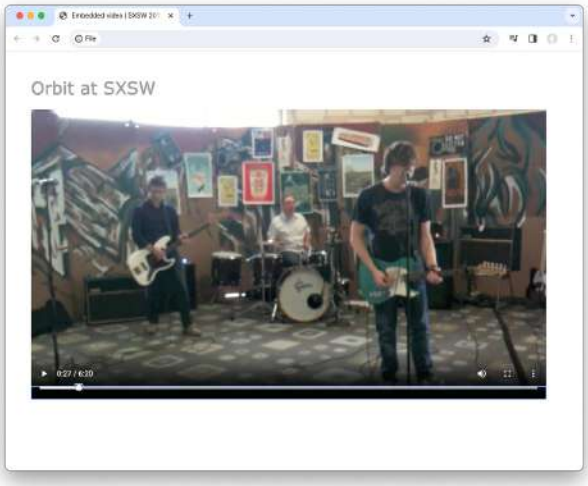


FIGURE 10-3. An embedded movie using the **video** element (shown in Chrome on macOS)

Here is a simple **video** element that embeds a movie and player in a web page:

```
<video src="highlight_reel.mp4" width="1280" height="720" →
poster="highlight_still.jpg" controls muted autoplay>
  Get the <a href="highlight_reel.mp4">MP4 video.</a>
</video>
```

Old browsers that do not support **video** display whatever content is provided within the **video** element. In this example, the content of the element provides a link to the movie that your visitor could download and play in another player.

In addition to **src**, there are several other attributes in this example that are worth looking at in detail:

width="*pixel measurement***"**
height="*pixel measurement***"**

Specifies the size of the box the embedded media player takes up on the screen. Generally, it is best to set the dimensions to exactly match the pixel dimensions of the movie (the **px** is not required because pixels are the default unit). The movie will resize to match the dimensions set here.

Streaming Media

Streaming allows users to view or listen to your content without actually downloading the media files. The files are stored remotely and delivered a few seconds at a time, on demand. Streaming makes sense for longer media, live events, and video conferencing.

There are a lot of players in the streaming media game. Some of the most popular formats include:

HLS (HTTP Live Streaming)

This format was developed by Apple in 2009, and it is the most popular streaming video format. It has the advantage of being able to be embedded with the **video** element. On the downside, it is not as efficient as newer streaming formats.

MPEG-DASH

MPEG-DASH offers an adaptive bitrate, which means it is always tailored to the user's internet connection speed. It is also compatible with most codecs. On the downside, it requires a custom video player, and browser support is lacking (it is not supported on Safari).

WebRTC

This streaming protocol is popular for live peer-to-peer streaming, such as video conferences.

`poster="url of image"`

Provides the location of an image that is shown in place of the video before it plays.

`controls`

Prompts the browser to display its built-in media controls, generally a play/pause button, a "seeker" that lets you move to a position within the video, and volume controls. It is possible to create your own custom player interface using CSS and JavaScript if you want more consistency across browsers.

`muted`

Plays the video track without the audio.

`autoplay`

Makes the video start playing automatically after the browser has downloaded enough of the media file to play through without stopping (see **Warning**).

WARNING

You should not assume a video will play automatically when you include the `autoplay` *attribute. Browsers like Chrome and Safari will only autoplay videos if the* `muted` *attribute is applied or if the video doesn't have an audio track at all.*

In addition, the `video` element can use the `loop` attribute to make the video play again after it has finished (ad infinitum). The `preload` attribute suggests to the browser whether it should fetch the video data as soon as the page loads (`preload="auto"`) or wait until the user clicks the play button (`preload="none"`). Setting `preload="metadata"` loads information about the media file, but not the media itself. A device can decide how to best handle the `auto` setting; for example, a browser on a smartphone may protect a user's data usage by not preloading media, even when it is set to `auto`. The `playsinline` attribute instructs the browser to display the video on the page rather than in a full-screen pop-up window. And finally, the `crossorigin` attribute allows authors to restrict how the element handles requests from other servers. Its values are `anonymous` and `use-credentials`.

Providing video format options

Do you remember back in **Chapter 7**, when we supplied multiple image formats with the `picture` element using a number of `source` elements? Well, `picture` got that idea from `video`!

Although we now have video formats that work reliably across browsers, there are always emerging codecs and containers that may do magical things with video compression but take a while to gain reliable support.

In the markup, a series of `source` elements inside the `video` element points to each video file with the `src` attribute (note that `src` is not provided in the `video` element itself). Browsers look down the list until they find one they support and download only that version. Emerging and cutting-edge formats should be listed higher in the list. The following example provides a video

clip in the souped-up WebM/AV1 format for supporting browsers, as well as an MP4 that will be used by browsers that don't play WebM. This will cover pretty much all browsers that support the video API:

```
<video id="video" controls poster="img/poster.jpg">
  <source src="clip.webm" type="video/webm">
  <source src="clip.mp4" type="video/mp4">
  <a href="clip.mp4">Download the MP4</a>
</video>
```

Custom video players

One of the powerful things about the **video** element and the **HTMLMediaElement** API is that the system allows for a lot of customization. You can change the appearance of the control buttons with CSS and manipulate the functionality with JavaScript. That's beyond the scope of this chapter, but here are two articles to get you started (JavaScript know-how required):

- "Creating a Cross-Browser Video Player" by Eric Shepherd, Chris Mills, and Ian Devlin (*oreil.ly/H1RFo*)
- "Build a Custom HTML5 Video Player with JavaScript" by Ifeoma Imoh (*oreil.ly/82gFf*)

You may also be interested in trying out a prefab video player like Video.js (*videojs.com*) that provides good looks and advanced performance, such as support for streaming video formats. A web search for "free HTML video player" will point you to plenty more options. You can implement many of them by adding a line or two of JavaScript to your document along with the **video** element, so it's not hard to get started.

Adding Audio to a Page

`<audio>...</audio>`
Adds an audio file to the page

If you've wrapped your head around the **video** markup example, you already know how to add audio to a page. The **audio** element uses the same attributes as the **video** element, with the exception of **width**, **height**, **playsinline**, and **poster** (because there is nothing to display). Just like with the **video** element, you can provide a stack of audio format options using the **source** element, as shown in the example here:

```
<p>Play "Percussion Gun" by White Rabbits</p>
<audio id="whiterabbits" controls preload="auto">
  <source src="percussiongun.mp3" type="audio/mp3">
  <source src="percussiongun.m4a" type="audio/mp4">
  <source src="percussiongun.webm" type="audio/webm">
  <p><a href="percussiongun.mp3">Download MP3</a></p>
</audio>
```

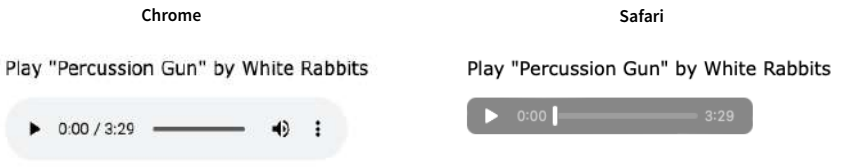


FIGURE 10-4. The audio player as rendered in Chrome and Safari

As FIGURE 10-4 shows, the audio player might look a little different from browser to browser. If you'd prefer consistency, you could create a custom player with JavaScript.

If you have only one audio file, you can simply use the **src** attribute instead. If you want to be evil, it is possible to embed audio in a page and set it to play automatically and then loop, without any controls to stop it, like this:

```
<audio src="jetfighter.mp3" autoplay loop></audio>
```

But you would never, *ever* do something like that, right? *Right?!* Of course you wouldn't.

Adding Text Tracks

`<track>...</track>`
Adds synchronized text to embedded media

The **track** element provides a way to add text that is synchronized with the timeline of a video or audio track. Some uses include the following:

- **Subtitles** in alternative languages
- **Captions** for the hearing impaired
- **Descriptions** of what is happening in a video for the sight impaired
- **Chapter titles** to allow for navigation through the media
- **Metadata** that is not displayed but can be used by scripts

Clearly, adding text tracks makes the media more accessible, but it has the added bonus of improving search engine optimization (SEO). It can also allow for **deep linking**, linking to a particular spot within the media's timeline.

FIGURE 10-5 shows how captions might be rendered in a browser.

> ■ **FURTHER READING**
>
> For more information, I recommend the MDN Web Docs article "Adding Captions and Subtitles to HTML5 Video" (*oreil.ly/IsL_9*).

FIGURE 10-5. A video with captions

Use the **track** element inside the **video** or **audio** element you wish to annotate. The **track** element must appear after all the **source** elements. The following code provides English and French subtitle options for a movie:

```
<video width="640" height="320" controls>
  <source src="japanese_movie.mp4" type="video/mp4">
  <source src="japanese_movie.webm" type="video/webm">
  <track src="english_subtitles.vtt"
         kind="subtitles"
         srclang="en"
         label="English subtitles"
         default>
  <track src="french.vtt"
         kind="subtitles"
         srclang="fr"
         label="Sous-titres en français">
</video>
```

The **track** element has the following attributes:

src

Points to the text file (*.vtt*) that contains the caption or subtitle text. WebVTT (Web Video Text Tracks) is a W3C standard that is designed to work well with the video API, including using CSS for styling and making the text searchable. The WebVTT standard is available at ***w3.org/TR/webvtt1/***. There are plenty of other options and services for adding captions and subtitles to videos, but they are outside of the HTML ecosystem.

kind

Specifies the type of text annotation you are providing (**subtitles**, **captions**, **descriptions**, **chapters**, or **metadata**). If **kind** is set to **subtitles**, you must also specify the language (in the **srclang** attribute) by using a standardized IANA two-letter language tag (see **Note**).

NOTE

The full list of two-letter language codes is published at ***oreil.ly/J6wa8****.*

`label`

Provides a name for the track that can be used in the interface for selecting a particular track.

`default`

Marks a particular track as the default. This attribute may be used on only one track within a media element.

In EXERCISE 10-2, you can spend some time playing around with the **video** element.

EXERCISE 10-2. Embedding a video player

In this exercise, you'll add a video to a page with the **video** element. In the materials for **Chapter 10**, you will find the small movie about wind tunnel testing that's used here, in MPEG-4 and WebM formats.

1. Create a new document with the proper HTML structure (or you can use the same document you used in EXERCISE 10-1).
2. The video in WebM format is smaller than the MP4 version, so we'll use the **video** element with multiple **source** options to serve the smaller WebM file to browsers that support it. Start by adding the **video** element, with **width** and **height** attributes that match the dimensions of the video (320 pixels wide by 262 pixels high). Add the **controls** attribute so you'll have a way to play and pause it. Finally, include a link to the MP4 file to serve as a fallback for browsers that don't recognize the **video** element (these days, there aren't many):

```
<video id="video" width="320" height="262" controls>
  <a href="windtunnel.mp4">Download the windtunnel video.</a>
</video>
```

3. Now add the **source** elements that provide the location and file type (MIME) information for both video formats. The video with the smallest file size (WebM) should go first:

```
<video id="video" width="320" height="262" controls >
  <source src="windtunnel.webm" type="video/webm">
  <source src="windtunnel.mp4" type="video/mp4">
  <a href="windtunnel.mp4">Download the windtunnel video.</a>
</video>
```

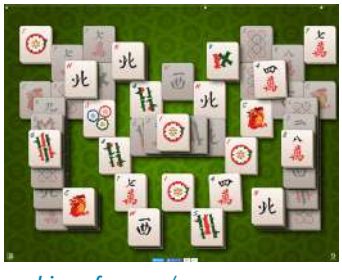
4. Save and view the document in your browser. If you see the fallback message, your browser doesn't support the **video** element. If you see the controls but no video, it doesn't support MP4. Chances are you'll see the video just fine and can play and pause it with the controls.
5. The **video** element is pretty straightforward, so you may feel like your job is done at this point, but I encourage you to play around with it a little to see what happens. Here are some things to try:
   - Resize the video player with the **width** and **height** attributes. What happens when the sizes you enter don't match the aspect ratio of the video? What if you set a value for **width** but omit the **height** attribute entirely?
   - Add the **autoplay** attribute.
   - Remove the **controls** attribute and see what that's like as a user.

CANVAS

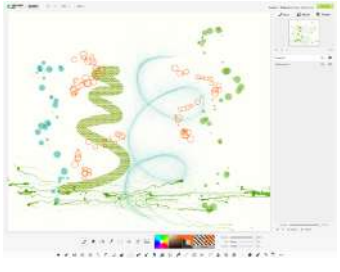
Another cool addition in HTML5 is the **canvas** element and the associated Canvas API. The **canvas** element creates a two-dimensional area on a web page for drawing with a set of JavaScript functions that create lines, shapes, fills, text, animations, and so on. What gives the **canvas** element so much potential is that it's all generated with scripting. That means it is dynamic and can draw things on the fly and respond to user input. This makes it a nifty platform for creating animations, games, and even whole applications.

It's worth noting that the canvas drawing area is raster-based, meaning that it is made up of a grid of pixels. This sets it apart from the other programmable drawing standard, SVG, which uses vector shapes and paths that are defined with points and mathematics. We'll talk about SVG more in **Chapter 28**.

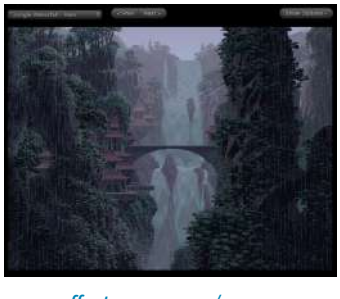
FIGURE 10-6 shows a few examples of the **canvas** element. You can find more examples at EnvatoTuts+ (*oreil.ly/efId40*). There are also many examples on CodePen (*codepen.io/tag/canvas*), where you can also inspect the JavaScript code that generates them.



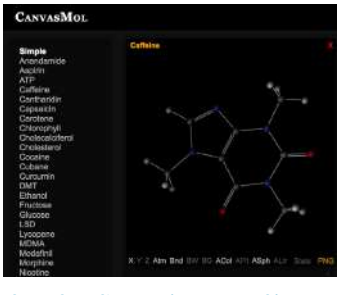
mahjong.frvr.com/



muro.deviantart.com/



www.effectgames.com/demoscanvascycle/



alteredqualia.com/canvasmol/

FIGURE 10-6. Examples of **canvas** used for games, animations, and applications

Mastering the **canvas** element is more than we can take on here, particularly without any JavaScript experience under our belts, but I'll give you a taste of what it is like to draw with JavaScript. That should give you a good idea of how it works, and also a new appreciation for the complexity of some of those examples.

The canvas Element

`<canvas>...</canvas>`
Adds a 2D dynamic drawing area

You add a canvas space to the page with the **canvas** element and specify the dimensions with the **width** and **height** attributes. You can also add an identifier with the **id** attribute. That's really all there is to the markup. For the rare browser that doesn't support the **canvas** element, you can provide some fallback content (a message, image, or whatever seems appropriate) inside the tags:

```
<canvas width="600" height="400" id="my_first_canvas">
  <img src="canvas_screenshot.png" alt="game interface">
</canvas>
```

The markup just clears a space upon which the drawing will happen. You can affect the drawing space itself with CSS (add a border or a background color, for example), but all of the contents of the canvas are generated by scripting and cannot be selected for styling with CSS.

Drawing with JavaScript

The Canvas API includes functions for creating shapes, such as **strokeRect()** for drawing a rectangular outline and **beginPath()** for starting a line drawing. Some functions move things around, such as **rotate()** and **scale()**. The API also includes attributes for applying styles (for example, **lineWidth**, **font**, **strokeStyle**, and **fillStyle**).

FIGURE 10-7. The finished product of the "Hello Canvas" example

Sanders Kleinfeld used the Canvas API to draw a simple smiley face drawing (FIGURE 10-7) for his book *HTML5 for Publishers* (O'Reilly). He was kind enough to allow me to use a sample of his code (available in full at *examples.oreilly.com/0636920022473/my_first_canvas/my_first_canvas.html*) in this book.

Here is a portion of the script that establishes the canvas, draws the border around it, draws the yellow circle, and writes the text at the bottom. If you would like to see the complete document (*canvas.html*), it's available with the exercise materials for this chapter. Don't worry that you don't know any JavaScript yet; just skim through the script and pay attention to the inline comments. I'll describe some of the functions that are used afterward, and I bet you'll get the gist of it just fine:

```
<script>
  var theCanvas = document.getElementById('my_first_canvas');
  var my_canvas = theCanvas.getContext('2d');

  my_canvas.strokeRect(0, 0, 200, 225) // to start, draw a border
around the canvas

  //draw face
  my_canvas.beginPath();
  my_canvas.arc(100, 100, 75, (Math.PI / 180) * 0, (Math.PI / 180) *
360, false); // circle dimensions
  my_canvas.strokeStyle = "black"; // circle outline is black
  my_canvas.lineWidth = 3; // outline is three pixels wide
```

```
    my_canvas.fillStyle = "yellow"; // fill circle with yellow
    my_canvas.stroke(); // draw circle
    my_canvas.fill(); // fill in circle
    my_canvas.closePath();

<-- script continues -->

    // Smiley Speaks!
    my_canvas.fillStyle = "black"; // switch to black for text fill
    my_canvas.font = '20px sans-serif'; // use 20 pixel sans serif font
    my_canvas.fillText("Hello Canvas!", 45, 200); // write text
    </script>
```

As promised, here is a little more information on a few of the Canvas API functions used in this example:

`strokeRect(`*`x1, y1, x2, y2`*`)`

Draws a rectangular outline from the point (*x1*, *y1*) to (*x2*, *y2*). By default, the origin of the canvas is the top-left corner, (0, 0), and *x* and *y* coordinates are measured to the right and down.

`beginPath()`

Starts a line drawing.

`closePath()`

Ends a line drawing that was started with **`beginPath()`**.

`stroke()`

Draws the line defined by the path. If you don't include this, the path won't appear on the canvas.

`fill()`

Fills in the path specified with **`beginPath()`** and **`endPath()`**.

`fillText(`*`your_text, x, y`*`)`

Adds text to the canvas starting at the (*x*, *y*) coordinate specified.

In addition, the following attributes were used to specify colors and styles:

`lineWidth`

Width of the border of the path

`strokeStyle`

Color of the border

`fillStyle`

Color of the fill (interior) of the shape created with the path

`font`

Font and size of the text

For more information about the **`canvas`** element and API, see the specification (***html.spec.whatwg.org/multipage/canvas.html***). A web search will turn up lots of Canvas tutorials and a few books, should you be ready to learn more.

TEST YOURSELF

We've looked at all sorts of ways to stick things in web pages in this chapter. You've seen how to use `iframe` to create a "window-in-a-window" for displaying external web resources, `object` for embedding media resources and video and audio players, and the `canvas` 2D scriptable drawing space. Now see if you were paying attention. As always, answers are in **Appendix A**.

1. What is a "nested browsing context," and how would you create one?

2. Why would you use the `sandbox` attribute with an `iframe` element?

3. Identify each of the following as a container format, video codec, or audio codec (hint: some have more than one answer):
   a. H.264
   b. VP9
   c. Vorbis
   d. WebM
   e. Opus
   f. MP4
   g. MP3

4. What does the `poster` attribute do?

5. What is a *.vtt* file?

6. What is the key difference between SVG and Canvas?

7. List the two Canvas API functions you would use to draw a rectangle and fill it with red. You don't need to write the whole script.

ELEMENT REVIEW: EMBEDDED MEDIA

The following elements are used to embed media files of many types into web pages.

TABLE 10-1. Embedded media elements

Element and attributes	Description
`audio`	Embeds an audio player in the page
`src="URL"`	The address of the resource
`preload="auto|none|metadata"`	Indicates how much the media resource should be buffered on page load
`autoplay`	Indicates the media can play as soon as the page is loaded
`loop`	Indicates the media file should start playing again automatically once it reaches the end
`muted`	Disables the audio output
`controls`	Indicates the browser should display a set of playback controls for the media file
`crossorigin`	Specifies how the element handles requests from other servers (CORS)
`canvas`	Represents a two-dimensional area that can be used for rendering dynamic bitmap graphics
`height`	The height of the canvas area
`width`	The width of the canvas area
`embed`	Embeds a multimedia object that requires a plug-in for playback on the page (note that certain media types require custom attributes not listed here)
`src="URL"`	The address of the media resource
`type="media type"`	The media (MIME) type of the media
`width="number"`	The horizontal dimension of the video player, in pixels
`height="number"`	The vertical dimension of the video player, in pixels
`iframe`	Creates a nested browsing context to display HTML resources in a page
`src="URL"`	The address of the HTML resource
`srcdoc="HTML source code"`	The HTML source of a document to display in the inline frame
`name="text"`	Assigns a name to the inline frame to be referenced by targeted links

`sandbox="`
`allow-forms|`
`allow-pointer-lock|`
`allow-popups|`
`allow-same-origin|`
`allow-scripts|`
`allow-top-navigation"`
	Security rules for nested content

`allow`	Applies a permissions policy to the iframe's contents
`allowfullscreen`	Indicates the objects in the inline frame are allowed to use **`requestFullScreen()`**
`referrerpolicy`	Identifies a referrer policy for fetches made by the element
`width="number"`	The horizontal dimension of the video player, in pixels
`height="number"`	The vertical dimension of the video player, in pixels

TABLE 10-1. Continued

Element and attributes	Description
`object`	A generic element for embedding an external resource
`data="URI"`	The address of the resource
`type="media type"`	The media (MIME) type of the resource
`name="text"`	Assigns a name to the object to be referenced by scripts
`form="form ID"`	Associates the **`object`** with a **`form`** element
`width="number"`	The horizontal dimension of the video player, in pixels
`height="number"`	The vertical dimension of the video player, in pixels
`source`	Allows authors to specify multiple versions of a media file (used with **`video`** and **`audio`**)
`src="text"`	The address of the resource
`type="media type"`	The media (MIME) type of the resource
`track`	Specifies an external resource (text or audio) that is timed with a media file that improves accessibility, navigation, or SEO
`kind="subtitles|captions|descriptions|chapters|metadata"`	The type of text track
`src="text"`	The address of the external resource
`srclang="valid language tag"`	The language of the text track
`label="text"`	A title for the track that may be displayed by the browser
`default`	Indicates the track should be used by default if it does not override user preferences
`video`	Embeds a video player in the page
`src="URL"`	The address of the resource
`poster="URL"`	The location of an image file that displays as a placeholder before the video begins to play
`preload="auto|none|metadata"`	Hints at how much buffering the media resource will need
`autoplay`	Indicates the media can play as soon as the page is loaded
`playsinline`	Encourages the browser to play video within the element's playback area
`crossorigin`	Specifies how the element handles requests from other servers (CORS)
`loop`	Indicates the media file should start playing again automatically once it reaches the end
`muted`	Disables the audio output
`controls`	Indicates the browser should display a set of playback controls for the media file
`width="number"`	Specifies the horizontal dimension of the video player, in pixels
`height="number"`	Specifies the vertical dimension of the video player, in pixels

III

CSS FOR PRESENTATION

CHAPTER
11

INTRODUCING CASCADING STYLE SHEETS

IN THIS CHAPTER

The benefits and power of CSS

How HTML markup creates a document structure

Writing style rules

Attaching styles to the HTML document

Inheritance, specificity, the cascade, rule order, and the box model

Units of measurement in CSS

Built-in browser tools for developers

I've talked about style sheets a fair bit bit already, and now we'll finally put them to work and start giving our pages some much-needed style. Cascading Style Sheets (CSS) is the W3C standard for defining the **presentation** of documents written in HTML (and XML languages as well). Presentation, again, refers to the way the document is delivered to the user, whether it's shown on a computer screen, displayed on a smartphone, printed on paper, or read aloud by a screen reader. With style sheets handling the presentation, HTML can handle the business of defining document structure and meaning, as intended.

"Presentation" is a pretty dry term for something that important and exciting. CSS is how an organization's brand shines through. It's what creates delightful user experiences. Color! Typography! Texture! CSS brings the party.

CSS is a separate language with its own syntax. This chapter covers CSS terminology and fundamental concepts that will help you get your bearings for the upcoming chapters, where you'll learn how to change text and font styles, add colors and backgrounds, and even do basic page layout. By the end of **Part III, CSS for Presentation**, you should have a solid foundation for further reading on your own and will have gotten lots of practice.

THE BENEFITS OF CSS

Not that you need further convincing, but here's a quick rundown of some of the benefits of using style sheets:

- **Less work.** You can change the appearance of an entire site by editing one style sheet. This also ensures consistency of formatting throughout the site.

- **More accessible sites.** When all matters of presentation are handled by CSS, you can mark up your content meaningfully, making it more accessible for screen readers and more findable for search engines.
- **Sophisticated designs and layouts.** Style sheets have evolved over the years to give us the ability to create precise grid-based layouts, advanced typography features, animation effects, and the ability to adapt those designs to various screen sizes and capabilities.

The Power of CSS

I'm not talking about minor visual tweaks here, like changing the color of headlines or adding text indents. When used to its full potential, CSS is a robust and powerful design tool. My eyes were first opened to the possibilities of using CSS for design by the variety and richness of the designs at the CSS Zen Garden (***csszengarden.com***).

In the misty days of yore (2003), when developers were still hesitant to give up their table-based layouts for CSS, David Shea's CSS Zen Garden site demonstrated exactly what could be accomplished using CSS alone. David posted an HTML document and invited designers to contribute their own style sheets that gave the document a visual design. FIGURE 11–1 shows just a few favorites. All of these designs use the *exact same* HTML source document.

Not only that, they don't include a single `img` element—all of the images are in element backgrounds. But look at how different each page looks—and how sophisticated. That's all done with style sheets. These examples are proof of the power in keeping presentation separate from structure.

The CSS Zen Garden got a code refresh in 2013, for its tenth anniversary, and David is keeping it alive as a learning tool for demonstrating what CSS can do. It is also a historical document of a turning point in the adoption of web standards. I encourage you pay it a visit and peruse some of the more recently added responsive designs.

Granted, it takes a lot of practice to be able to create CSS layouts like those shown in FIGURE 11–1. Killer graphic design skills help too (unfortunately, you won't learn those in this book). I'm showing this to you up front because I want you to be aware of the potential of CSS-based design, particularly because the examples in this beginners' book tend to be very simple and straightforward. Take your time learning, but keep your eye on the prize.

HOW STYLE SHEETS WORK

It's as easy as 1-2-3!

1. Start with a document that has been marked up in HTML.
2. Write style rules for how you'd like certain elements to look.

CSS Zen Dragen
by Matthew Buchanan

By the Pier
by Peter Ong Kelmscott

Organica Creativa
by Eduardo Cesario

Shaolin Yokobue
by Javier Cabrera

FIGURE 11-1. These pages from the CSS Zen Garden use the same HTML source document, but the design is changed with CSS alone (used with permission of CSS Zen Garden and the individual designers).

3. Attach the style rules to the document (there are three ways to do this, as you'll see shortly). When the browser displays the document, it follows your rules for rendering elements.

OK, so there's a bit more to it than that, of course. Let's give each of these steps some more consideration.

1. Marking Up the Document

You know a lot about marking up content from the previous chapters. For example, you know that it is important to choose elements that accurately describe the meaning of the content. You also know that the markup creates the structure of the document, sometimes called the **structural layer**, upon which the **presentation layer** can be applied.

EXERCISE 11-1. A first look

In this chapter, we'll add a few simple styles to a short article. The document, *cooking.html*, and its associated image, *salads.jpg*, are available at *learningwebdesign.com/6e/materials*.

For now, just open the document in a browser to see how it looks by default (it should look something like FIGURE 11-2). You can also open the document in a text or code editor to get ready to follow along in the next two exercises.

Cooking with Daniel from Nada Surf

Six-hour Salad

The Main Course

FIGURE 11-2. This is what the article looks like without any style sheet instructions. Although we won't be making it beautiful, you will get a feel for how style sheets work.

In this and the upcoming chapters, you'll see that having an understanding of your document's structure and the relationships between elements is central to your work as a style sheet author.

In the exercises throughout this chapter you will get a feel for how simple it is to change the look of a document with style sheets. The good news is that I've whipped up a little HTML document for you to play with. You can get acquainted with the page we'll be working with in EXERCISE 11-1.

2. Writing the Rules

A style sheet is made up of one or more instructions (called **style rules**) that describe how an element or group of elements should be displayed. The first step in learning CSS is to get familiar with the parts of a rule. As you'll see, they're fairly intuitive to follow. Each rule *selects* an element and *declares* how it should look.

The following example contains two rules. The first makes all the `h1` elements in the document green; the second specifies that the paragraphs should be in a large, sans-serif font. **Sans-serif** fonts do not have a little slab (a **serif**) at the ends of strokes and tend to look more sleek and modern.

```
h1 { color: green; }
p  { font-size: large; font-family: sans-serif; }
```

In CSS terminology, the two main sections of a rule are the **selector** that identifies the element or elements to be affected, and the **declaration** that provides the rendering instructions. The declaration, in turn, is made up of a **property** (such as `color`) and its **value** (`green`), separated by a colon and a space. One or more declarations are placed inside curly braces, as shown in FIGURE 11-3.

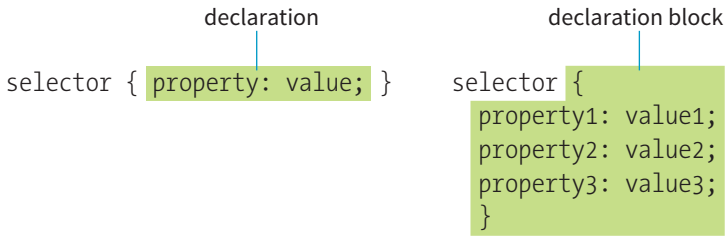


FIGURE 11-3. The parts of a style rule

Selectors

In the previous small style sheet example, the `h1` and `p` elements are used as selectors. This is called an **element type selector**, and it is the most basic type of selector. The properties defined for each rule will apply to every `h1` and `p` element in the document, respectively.

Another type of selector is an **ID selector**, which selects an element based on the value of an element's `id` attribute. It is indicated with the `#` symbol. For example, the selector `#recipe` targets an element with `id="recipe"`. I'll go into more detail about this selector in the next chapter.

In upcoming chapters, I'll also introduce you to some more sophisticated selectors that you can use to target elements, including ways to select groups of elements and elements that appear in a particular context. See the **"Selectors in This Book"** sidebar for details.

Mastering selectors—that is, choosing the best type of selector and using it strategically—is an important step in mastering CSS.

Selectors in This Book

Instead of throwing the selectors at you all at once, I've spread them out so you can master a few at a time. Here is where you will find them:

Chapter 11, Introducing Cascading Style Sheets:

- Element type selector (p.228)
- Grouped selectors (p.240)

Chapter 12, Formatting Text:

- Descendant selectors (p.268)
- Child, next-sibling, and following-sibling selectors (p.270)
- ID and class selectors (p.271)
- Universal selector (*) (p.272)

Chapter 14, Selectors:

- Summary of previous selector types (p.329)
- Attribute selectors (p.330)
- Pseudo-class selectors (p.332)
- Pseudo-element selectors (p.348)

In addition, you can find all of the available selectors in **Appendix C, CSS Selectors Level 4**.

Declarations

A declaration is made up of a property/value pair. There can be more than one declaration in a single rule; for example, the rule for the `p` element shown earlier has declarations for both the `font-size` and `font-family` properties. Each declaration must end with a semicolon to keep it separate from the following declaration (see **Note**). If you omit the semicolon, the declaration and the one following it will be ignored. The curly braces and the declarations they contain are often referred to as the **declaration block** (FIGURE 11-3).

Because CSS ignores whitespace and line returns within the declaration block, authors typically write each declaration in the block on its own line, as shown in the following example. This makes it easier to find the properties applied to the selector and to tell when the style rule ends:

```
h1 {
  color: green;
}
p {
  font-size: large;
  font-family: sans-serif;
}
```

Note that nothing has really changed here—there is still one set of curly braces, a semicolon after each declaration, and so on. The only difference is the insertion of line returns and some character spaces for alignment.

NOTE

Technically, the semicolon is not required after the last declaration in the block, but it is recommended that you get into the habit of always ending declarations with a semicolon. This will make adding declarations to the rule later that much easier.

Properties

The heart of style sheets lies in the collection of standard properties that can be applied to selected elements. The CSS specification defines hundreds of properties, for everything from text indents to how table headers should be read aloud. This book covers the most common and best-supported properties that you can begin using right away. If you are curious, you can see the full list of properties at ***w3.org/TR/css/#properties***.

Values

Values are dependent on the property. Some properties take length measurements, some take color values, and others have a predefined list of keywords. When you use a property, it is important to know which values it accepts; however, in many cases, simple common sense will serve you well. Some code-editing tools provide hints of suitable values to choose from when you type the property name. Before we move on, why not get a little practice writing style rules yourself in EXERCISE 11-2?

3. Attaching the Styles to the Document

In the previous exercise, we embedded the style sheet right in the document by using the **style** element. That is just one of three ways that style information can be applied to an HTML document. You'll get to try out the others soon, but it's helpful to have an overview of the methods and terminology up front:

External style sheets

An external style sheet is a separate, text-only document that contains a number of style rules. It should be named with the *.css* suffix. The *.css* document is then linked to (via the **link** element) or imported into (via an **@import** rule in a style sheet) one or more HTML documents. In this way, all the files in a website may share the same style sheet. This is the most powerful and preferred method for attaching style sheets to content. We'll discuss external style sheets and start using them in **Chapter 14, Selectors**. In the meantime, here is a quick look at the syntax for the **link** element used for style sheet attachment:

```
<head>
  <link rel="stylesheet" href="/path/stylesheet.css">
</head>
```

Embedded style sheets

This is the type of style sheet we worked with in EXERCISE 11-2. It is placed in a document via the **style** element, and its rules apply only to that document. The **style** element must be placed in the **head** of the document. This example also includes a comment (see the **"Comments in Style Sheets"** sidebar):

```
<head>
   <title>Document Title</title>
   <style>
     /* style rules go here */
   </style>
</head>
```

Comments in Style Sheets

Sometimes it is helpful to leave yourself or your collaborators comments in a style sheet. CSS has its own comment syntax, shown here:

```
/* comment goes here */
```

Content between the /* and */ will be ignored when the style sheet is parsed, which means you can leave comments anywhere in a style sheet—even within a rule:

```
body {
  font-size: small;
 /* change this later */
}
```

One use for comments is to label sections of the style sheet to make things easier to find later. For example:

```
/* FOOTER STYLES */
```

CSS comments are also useful for temporarily hiding style declarations in the design process. When you are trying out different styles, you can quickly switch them off by enclosing them in /* and */, check the design in a browser, then remove the comment characters to make the style appear again. This is much faster than deleting and retyping the declarations.

EXERCISE 11-2. Your first style sheet

FIGURE 11-2 showed how the cooking article used in this exercise looks with the browser's default styles. Open the file *cooking.html* in a text or code editor, and we'll get to work adding our own. In the **head** of the document, you will find that I have set up a **style** element for you to type the rules into. The **style** element is used to embed a style sheet right in the HTML document.

To begin, we'll simply add the small style sheet we've been using in this section. Type the following rules into the document, just as you see them here:

```
<style>
  h1 {
    color: green;
  }
  p {
    font-size: large;
    font-family: sans-serif;
  }
</style>
```

Save the file, and take a look at it in the browser (FIGURE 11-4, left). You should notice some minor changes (if your browser already uses a sans-serif font, you may see only a size change in the paragraphs). If not, go back and check that you included both the opening and closing curly braces and semicolons. It's easy to omit these characters, causing the style sheet to not work.

Now we'll edit the style sheet to see how easy it is to write rules and see the effects of the changes. Here are a few things to try.

IMPORTANT: Remember that you need to save the document after each change in order for the changes to be visible when you reload it in the browser.

- Make the **h1** element "gray" and take a look at it in the browser. Then make it "blue". Finally, make it "maroon". (I'll run through the complete list of available color names in **Chapter 13, Colors and Backgrounds**.)
- Add a new rule that makes the **h2** elements maroon as well.
- Add a 100-pixel left margin to paragraph (**p**) elements by using the following declaration. Margins add space around an element, and in this example, we are adding space to the left side only:

  ```
  margin-left: 100px;
  ```

  Remember that you can add this new declaration to the existing rule for **p** elements.
- Create a new rule to add a 100-pixel left margin to the **h2** headings as well.
- Now, we'll add a 1-pixel solid maroon line above and below the **h1** element with **border-** properties and give it a little breathing room with **padding**. Here are the declarations to add to the existing **h1** rule:

  ```
  border-bottom: 1px solid maroon;
  border-top: 1px solid maroon;
  padding: 12px;
  ```
- Move the image to the right and allow text to flow around it with the **float** property. The shorthand **margin** property shown in this rule adds 0 pixels of space on the top and bottom of the image, and 12 pixels of space on the left and right of the image (the values are mirrored in a manner explained in **Chapter 15, Thinking Inside the Box**):

  ```
  img {
    float: right;
    margin: 0 12px;
  }
  ```

When you are done, the document should look something like the one shown in FIGURE 11-4 (right). Not beautiful—just different.

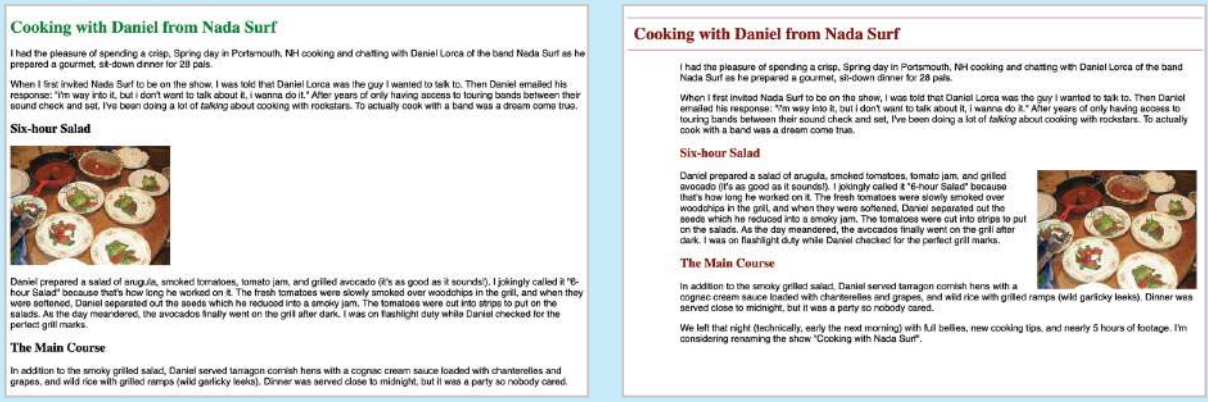

Cooking with Daniel from Nada Surf
I had the pleasure of spending a crisp, Spring day in Portsmouth, NH cooking and chatting with Daniel Lorca of the band Nada Surf as he prepared a gourmet, sit-down dinner for 28 pals.
When I first invited Nada Surf to be on the show, I was told that Daniel Lorca was the guy I wanted to talk to. Then Daniel emailed his response: "I'm way into it, but I don't want to talk about it, I wanna do it." After years of only having access to touring bands between their sound check and set, I've been doing a lot of talking about cooking with rockstars. To actually cook with a band was a dream come true.
Six-hour Salad
Daniel prepared a salad of arugula, smoked tomatoes, tomato jam, and grilled avocado (it's as good as it sounds!). I jokingly called it "6-hour Salad" because that's how long he worked on it. The fresh tomatoes were slowly smoked over woodchips in the grill, and when they were softened, Daniel separated out the seeds which he reduced into a smoky jam. The tomatoes were cut into strips to put on the salads. As the day meandered, the avocados finally went on the grill after dark. I was on flashlight duty while Daniel checked for the perfect grill marks.
The Main Course
In addition to the smoky grilled salad, Daniel served tarragon cornish hens with a cognac cream sauce loaded with chanterelles and grapes, and wild rice with grilled ramps (wild garlicky leeks). Dinner was served close to midnight, but it was a party so nobody cared.
We left that night (technically, early the next morning) with full bellies, new cooking tips, and nearly 5 hours of footage. I'm considering renaming the show "Cooking with Nada Surf".

FIGURE 11-4. The article after we added the first two style rules (left) and the final document at the end of the exercise (right)

EXERCISE 11-3. Applying an inline style

Open the article *cooking.html*, in whatever state you last left it in EXERCISE 11-2. If you worked to the end of the exercise, you will have a rule that makes the **h2** elements maroon.

Let's write an inline style that makes the second **h2** orange. We'll do that right in the opening **h2** tag by using the **style** attribute, as shown here:

```
<h2 style="color: orange;">The
Main Course</h2>
```

Save the file and open it in a browser. Now the second heading is orange, overriding the maroon color set in the embedded style sheet. The other **h2** heading is unaffected.

Inline styles

You can apply properties and values to a single element by using the **style** *attribute* in the element itself, as shown here:

```
<h1 style="color: red;">Introduction</h1>
```

To add multiple properties, just separate them with semicolons, like this:

```
<h1 style="color: red; margin-top: 2em;">Introduction</h1>
```

Inline styles apply only to the particular element in which they appear. They should generally be avoided, unless it is necessary to override styles from an embedded or external style sheet. Inline styles are problematic in that they intersperse presentation information into the structural markup. They also make it more difficult to make changes, because every **style** attribute must be hunted down and edited in the HTML source. Use them sparingly and with caution (see **Note**).

NOTE

Despite these drawbacks, some web development frameworks, like Tailwind, rely heavily on inline styles, effectively baking the styles into the markup. The price of convenience, I suppose.

EXERCISE 11-3 gives you an opportunity to write an inline style and see how it works. We won't be working with inline styles after this point, for the reasons listed earlier, so here's your chance.

THE BIG CONCEPTS

There are a few big ideas that you need to be familiar with to be comfortable with how Cascading Style Sheets behave. I'm going to introduce you to these concepts now so I don't have to slow down for a lecture once we're rolling through the style properties. Each of these ideas will be revisited and illustrated in more detail in the upcoming chapters.

Inheritance

Are your eyes the same color as your parents'? Did you inherit their hair color? Well, just as parents pass down traits to their children, styled HTML elements pass down certain style properties to the elements they contain. Notice in EXERCISE 11-2, when we styled the **p** elements in a large, sans-serif font, the **em** element in the second paragraph became large and sans-serif as well, even though we didn't write a rule for it specifically (FIGURE 11-5). That is because the **em** element **inherited** the styles from the paragraph it is in. Inheritance provides a mechanism for styling elements that don't have any explicit styles rules of their own.

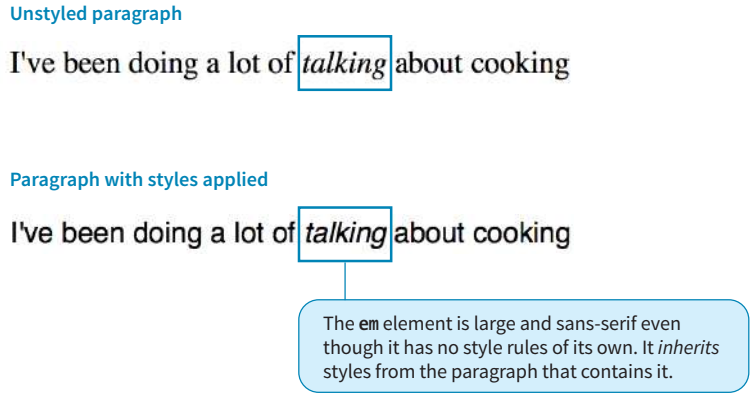

Unstyled paragraph
I've been doing a lot of talking about cooking
Paragraph with styles applied
I've been doing a lot of talking about cooking
The em element is large and sans-serif even though it has no style rules of its own. It inherits styles from the paragraph that contains it.

FIGURE 11-5. The `em` element inherits styles that were applied to the paragraph.

Document structure

This is where an understanding of your document's structure becomes important. As I've noted before, HTML documents have an implicit structure, or hierarchy, called the Document Object Model (DOM). For example, the sample article we've been playing with has an **`html`** root element that contains a **`head`** and a **`body`**, and the **`body`** contains heading and paragraph elements. A few of the paragraphs, in turn, contain inline elements such as images (**`img`**) and emphasized text (**`em`**). You can visualize the structure as an upside-down tree, branching out from the root, as shown in FIGURE 11-6.

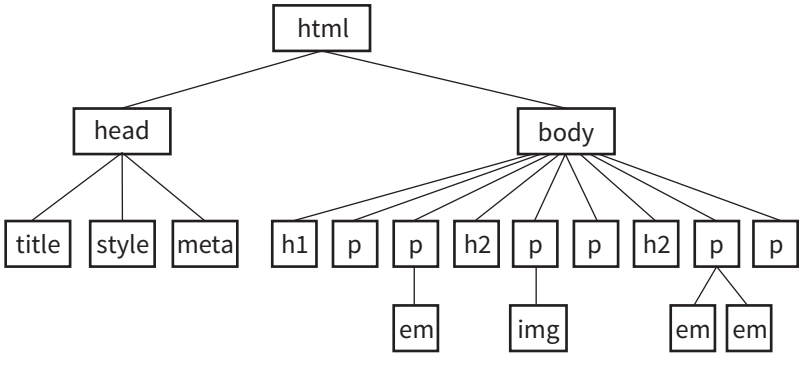


FIGURE 11-6. The document tree structure of the sample document, *cooking.html*

Parents and children

The document tree becomes a family tree when it comes to referring to the relationships between elements. All the elements contained within a given element are said to be its **descendants**. For example, the **`h1`**, **`h2`**, **`p`**, **`em`**, and **`img`** elements in the document in FIGURE 11-6 are all descendants of the **`body`** element.

An element that is directly contained within another element (with no intervening hierarchical levels) is said to be the **child** of that element. Conversely, the containing element is its **parent**. For example, the `em` element is the child of the `p` element, and the `p` element is its parent.

All of the elements higher than a particular element in the hierarchy are its **ancestors**. Two elements with the same parent are **siblings**. We don't refer to "aunts" or "cousins," so the analogy stops there. This may all seem academic, but it will come in handy when you're writing CSS selectors.

Pass it on

When you write a font-related style rule using the `p` element as a selector, the rule applies to all of the paragraphs in the document as well as the inline text elements they contain. We saw evidence of the `em` element inheriting the style properties applied to its parent (`p`) back in FIGURE 11-5. FIGURE 11-7 demonstrates what's happening in terms of the document structure diagram. As you can see here, the `img` element is excluded because font-related properties do not apply to images.

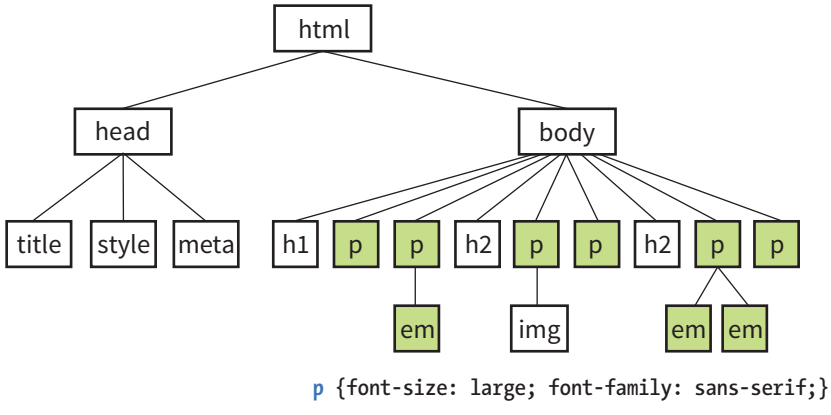


FIGURE 11-7. Certain properties applied to the `p` element are inherited by their children.

CSS TIP

When you learn about a new property, it's a good idea to note whether it inherits. Inheritance is indicated for every property listing in this book. For the most part, inheritance follows your expectations, but the CSS spec (*w3.org/TR/CSS*) is the definitive source.

Notice that I've been saying "certain" properties are inherited. It's important to note that some style sheet properties inherit and others do not. In general, properties related to the styling of text—font size, color, style, and the like—are passed down. Properties such as borders, margins, backgrounds, and so on that affect the boxed area around the element tend not to be passed down. This makes sense when you think about it. For example, if you put a border around a paragraph, you wouldn't want a border around every inline element (such as `em`, `strong`, or `a`) it contains as well.

You can use inheritance to your advantage when writing style sheets. For example, if you wanted all text elements to be blue, you could write separate style rules for every element in the document and set the **color** to blue. But a *better* way would be to write a single style rule that applies the **color** property to the **body** element and let all the elements contained in the **body** inherit that style (FIGURE 11-8).

Any property applied to a specific element overrides the inherited values for that property. Going back to the article example, if we specify that the **em** element should be orange, that will override the inherited blue setting.

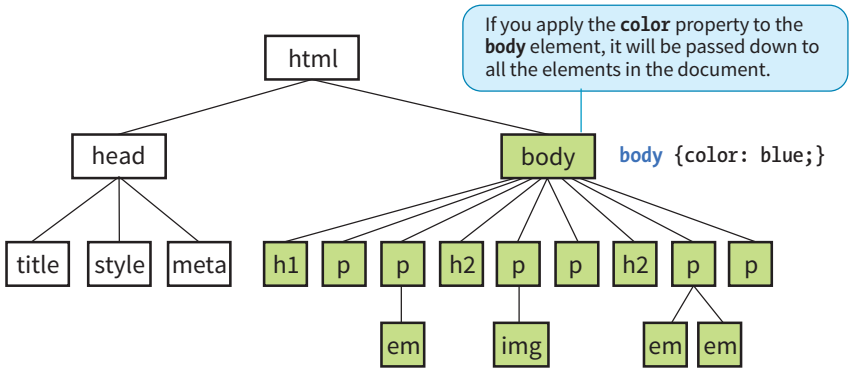


FIGURE 11-8. All the elements in the document inherit certain properties applied to the body element. Deliberately applied styles override inherited styles.

Conflicting Styles: The Cascade

Ever wonder why they are called "cascading" style sheets? CSS allows you to apply several style sheets to the same document, which means there are bound to be conflicts. For example, what should the browser do if a document's imported style sheet says that **h1** elements should be red, but its embedded style sheet has a rule that makes **h1**s purple? The two style rules with **h1** selectors have equal weight, right?

The folks who wrote the style sheet specification anticipated this problem and devised a hierarchical system that assigns different weights to the various sources of style information. The **cascade** refers to what happens when several sources of style information vie for control of the elements on a page: style information is passed down ("cascades" down) until it is overridden by a style rule with more weight. Weight is considered based on the *priority* of the style rule source, the *specificity* of the selector, and *rule order*.

The "cascade" refers to what happens when several sources of style information vie for control of the elements on a page.

Priority

If you don't apply any style information to a web page, it renders according to the browser's internal style sheet. I've been calling this the default rendering; the W3C calls it the **user agent style sheet**. Individual users can apply their own styles as well (the **user style sheet**, also called the **reader style sheet**), which override the default styles in their browser. However, if the author of the web page has attached a style sheet (the **author style sheet**), that one overrides both the user and the user agent styles. The sidebar **"Style Rule Hierarchy"** provides an overview of the cascading order from highest to lowest priority.

The only exception is if the user has identified a style as "important," in which case that style will override all competing styles (see the **"Assigning Importance"** sidebar). This permits users to maintain settings accommodating a disability, such as extra-large type for sight impairment.

Style Rule Hierarchy

Style information can come from various origins, and there is a strict order of precedence. Here's an overview—items higher in the list override items lower down:

- Style rules marked `!important` by the user agent (browser or alternative device)
- Any style rule marked `!important` by the reader (user)
- Any style rule marked `!important` by the author of the HTML document
- Style sheets written by the author of the document
- Style sheets created by the reader (user)
- The browser's default style rules ("user agent style sheet")

There are a few other style origins in the current spec related to animation that I haven't included here, because I don't want to get ahead of myself. You can read the details about the cascade sorting order at *w3.org/TR/css-cascade-5/#cascade-sort*.

Assigning Importance

If you want a rule not to be overridden by a subsequent conflicting rule, include the `!important` indicator just after the property value and before the semicolon for that rule. For example, to guarantee paragraph text will be blue, use the following rule:

```
p {color: blue !important;}
```

Even if the browser encounters an inline style later in the document (which should override a document-wide style sheet), like this one:

```
<p style="color: red">
```

that paragraph will still be blue because the rule with the `!important` indicator cannot be overridden by other styles in the author's style sheet.

The only way an `!important` rule may be overridden is by a conflicting rule in a reader (user) or user agent (browser) style sheet that has also been marked `!important`. This is to ensure that special reader requirements, such as large type or high-contrast text for the visually impaired, are never overridden.

Based on the previous examples, if the reader's style sheet includes this rule:

```
p {color: black;}
```

the text will still be blue because all author styles (even those not marked `!important`) take precedence over the reader's styles. However, if the conflicting reader's style is marked `!important`, like this:

```
p {color: black !important;}
```

the paragraphs will be black, and no author-provided style will be able to override this.

Beware that the `!important` indicator is not a get-out-of-jail-free card. Best practice dictates that it be used sparingly, if at all, and certainly never just to get yourself out of a sticky situation with inheritance and the cascade.

Specificity

When two rules in a single style sheet conflict, the type of selector is used to determine the winner.

It is possible for conflicts to arise in which an element is getting style instructions from more than one rule. In these cases, rules with more specific selectors have more "weight" and will override conflicting declarations. This is best explained with an example.

In the following style sheet, the first rule sets the color of all paragraphs to green, but there is another rule that makes an "intro" paragraph purple. Which rule should the "intro" paragraph use?

THE MARKUP

```
<p id="intro">Lorem ipsum odor amet, consectetuer adipiscing elit.
Libero...</p>
<p>Amet curae sagittis...</p>
<p>Risus urna egestas...</p>
```

THE STYLES

```
p { color: green; }
p#intro { color: purple; }
```

In this example, the rule that includes the ID selector (**`#intro`**) wins because ID selectors are more specific than general element selectors (like **`p`**). The "intro" paragraph will be purple, overriding the green style applied to all paragraphs (FIGURE 11-9).

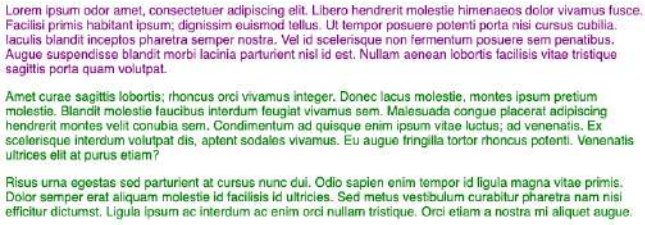

Lorem ipsum odor amet, consectetuer adipiscing elit. Libero hendrerit molestie himenaeos dolor vivamus fusce. Facilisi primis habitant ipsum; dignissim euismod tellus. Ut tempor posuere potenti porta nisi cursus cubilia. Iaculis blandit inceptos pharetra semper nostra. Vel id scelerisque non fermentum posuere sem penatibus. Augue suspendisse blandit morbi lacinia parturient nisl id est. Nullam aenean lobortis facilisis vitae tristique sagittis porta quam volutpat.
Amet curae sagittis lobortis; rhoncus orci vivamus integer. Donec lacus molestie, montes ipsum pretium molestie. Blandit molestie faucibus interdum feugiat vivamus sem. Malesuada congue placerat adipiscing hendrerit montes velit conubia sem. Condimentum ad quisque enim ipsum vitae luctus; ad venenatis. Ex scelerisque interdum volutpat dis, aptent sodales vivamus. Eu augue fringilla tortor rhoncus potenti. Venenatis ultrices elit at purus etiam?
Risus urna egestas sed parturient at cursus nunc dui. Odio sapien enim tempor id ligula magna vitae primis. Dolor semper erat aliquam molestie id facilisis id ultricies. Sed metus vestibulum curabitur pharetra nam nisi efficitur dictumst. Ligula ipsum ac interdum ac enim orci nullam tristique. Orci etiam a nostra mi aliquet augue.

FIGURE 11-9. The paragraph labeled "intro" displays in purple because the style rule with an `id` value is more specific (and therefore has more weight) than a simple element selector rule (`p`).

It's a little soon to be discussing specificity, because we've only looked at a few selector types. For now, put the term **specificity** and the concept that some selectors have more "weight," and therefore override others, on your radar. We'll revisit specificity in much more detail in **Chapter 14, Selectors**, when you have more selector types under your belt.

Rule order

After all the style sheet sources have been sorted by priority and all the linked and imported style sheets have been shuffled into place, there are still likely to be conflicts in rules with equal weights. When these conflicts arise, the order in which the rules appear is important. The cascade follows a "last one wins" rule. Whichever rule appears last has the last word.

Suppose, for example, that you have these three rules in your style sheet:

```
<style>
  p { color: red; }
  p { color: blue; }
  p { color: green; }
</style>
```

In this scenario, paragraph text will be green, because the last rule in the style sheet overrides the earlier ones. Procedurally, the paragraph is assigned one color, then another, and finally a third one (green), which is the color that stays. The same thing happens when conflicting styles occur within a single declaration stack:

```
<style>
  p {
    color: red;
    color: blue;
    color: green;
  }
</style>
```

The resulting color will be green, because the last declaration overrides the previous two. It's easy to accidentally override previous declarations within a rule when you get into shorthand properties (which I'll introduce in **Chapter 12**), so this is an important behavior to keep in mind. The example we just looked at was a very simple one. What happens when style sheet rules from different sources come into play? Let's take a look.

Putting it together

Behind the scenes, the browser may be juggling conflicting rules from multiple sources and in different orders. Let's consider an HTML document that has an embedded style sheet (added with the **`style`** element) that starts with an **`@import`** rule for importing an external *.css* file. That same HTML document also has a few inline **`style`** attributes applied to particular **`h1`** elements.

STYLE DOCUMENT (*external.css*):

```
...
h1 { color: red }
...
```

HTML DOCUMENT:

```
<!DOCTYPE html>
<html>
<head>
  <title>...</title>
  <style>
    @import url(external.css);  /* set to red first */
    h1 { color: purple;}  /* overridden by purple */
  </style>
</head>
<body>
```

```
    <h1 style="color: blue">Heading</h1>
    /* inline styles have more priority and win */
    ...
</body>
</html>
```

When the browser parses the file, it gets to the imported style sheet first, which sets **h1**s to red. Then it finds a rule with equal weight in the embedded style sheet that overrides the imported rule, setting **h1**s to purple. As it continues, it encounters a style rule right in an **h1** that sets its color to blue. Because that rule is in an inline style, it takes priority, so that particular **h1** will be blue (that's the effect we witnessed in EXERCISE 11-3). The other **h1**s in this document without inline style rules will be purple, because that was the last **h1** color applied to the whole document.

The Box Model

As long as we're talking about Big CSS Concepts, it's only appropriate to introduce the cornerstone of the CSS visual formatting system: the **box model**. The easiest way to think of the box model is that browsers see every element on the page (both block and inline) as being contained in a little rectangular box. You can apply properties such as borders, margins, padding, and backgrounds to these boxes, and even reposition them on the page.

To see the elements roughly the way the browser sees them, I've written style rules that add borders around every content element in our sample article:

```
h1 { border: 1px solid blue; }
h2 { border: 1px solid blue; }
p { border: 1px solid blue; }
em { border: 1px solid blue; }
img { border: 1px solid blue; }
```

FIGURE 11-10 shows the results. The borders reveal the shape of each block element box. There are boxes around the inline elements (**em** and **img**) as well. If you look at the headings, you will see that block element boxes expand to fill the available width of the browser window, which is the nature of block elements in the normal document flow. Inline boxes encompass just the characters or images they contain.

The full story is that element boxes are made up of content and some amount of padding (space around the content), a border, and a margin (space outside the border), as shown as FIGURE 11-11. I'm going to go into a *lot* more detail about the box model in **Chapter 15, Thinking Inside the Box**, but having this basic mental model will benefit you as we discuss text and backgrounds in the following two chapters.

Using Rule Order for Fallbacks

Many CSS properties are tried and true and are supported by all browsers. However, there are always useful new properties emerging that take a while to be implemented by browsers. It's common for just one or two browsers to support a new feature and for others to lag behind, or never support it at all.

Fortunately, there are a number of ways to provide **fallbacks** (alternative styles using better-supported properties) to non-supporting browsers. The most straightforward method takes advantage of browsers' built-in behavior of ignoring any declaration they don't understand and then using rule order strategically.

In this example, I want **div** elements to use a theoretical, cutting-edge property, **texture** (which I've just made up for this example). I know not many browsers support textures, so I provide a fallback style (a yellow **background-color**) to use if the texture isn't applied. Supporting browsers display the bumpy texture (again, totally made up!), because it is the last rule in the stack. Non-supporting browsers make the background yellow, and effectively stop there. When they get to the **texture** property, which they don't understand, they don't crash or throw an error; they just ignore it.

```
div {
  background-color: yellow;
  /* fallback first */
  texture: bumpy;
  /* cutting-edge property
(theoretical) */
}
```

A Quick History of CSS

The first official version of CSS (the **CSS Level 1 Recommendation**, a.k.a. **CSS1**), released in 1996, included properties for adding font, color, and spacing instructions to page elements. Unfortunately, lack of browser support prevented the widespread adoption of CSS for several years.

CSS Level 2 (CSS2), released in 1998, most notably added properties for positioning that allowed CSS to be used for page layout. It also introduced styles for other media types (such as print and handheld) and more sophisticated methods for selecting elements. **CSS Level 2 Revision 1 (CSS2.1)** made some minor adjustments to CSS2 and became a recommendation in 2011.

CSS2 was the last numbered version that encompassed all CSS properties. After that, the spec was divided into individual modules, each addressing a feature, such as animation, multiple column layouts, or borders. Each module has its own numbering system indicating the development of that particular feature. Some modules have become standardized at Level 3. As of this writing, others have achieved Level 4 or even Level 5 status, although those later levels tend to include experimental features that are not well supported. With each module on its own track, browser developers can begin implementing (and we can begin using!) one feature at a time instead of waiting for an entire specification version to be "ready."

You won't believe how many individual specifications are in the works! For an overview of the specifications in their various states of "doneness," see the W3C's CSS current work page at *w3.org/Style/CSS/current-work*.

Grouped Selectors

Hey! This is a good opportunity to show you a handy style rule shortcut. If you ever need to apply the same style property to a number of elements, you can group the selectors into one rule by separating them with commas. This one rule has the same effect as the five rules listed in the previous section. Grouping them makes future edits more efficient and results in a slightly smaller file size:

```
h1, h2, p, em, img { border: 1px solid blue; }
```

Now you have two selector types in your toolbox: the simple element selector and grouped selectors.

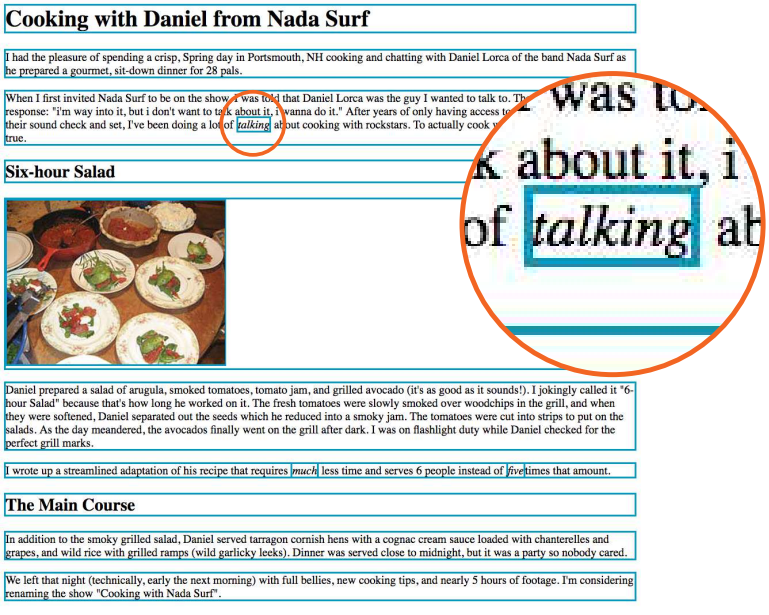

Cooking with Daniel from Nada Surf
I had the pleasure of spending a crisp, Spring day in Portsmouth, NH cooking and chatting with Daniel Lorca of the band Nada Surf as he prepared a gourmet, sit-down dinner for 28 pals.
talking
Six-hour Salad
Daniel prepared a salad of arugula, smoked tomatoes, tomato jam, and grilled avocado (it's as good as it sounds!). I jokingly called it "6-hour Salad" because that's how long he worked on it. The fresh tomatoes were slowly smoked over woodchips in the grill, and when they were softened, Daniel separated out the seeds which he reduced into a smoky jam. The tomatoes were cut into strips to put on the salads. As the day meandered, the avocados finally went on the grill after dark. I was on flashlight duty while Daniel checked for the perfect grill marks.
I wrote up a streamlined adaptation of his recipe that requires much less time and serves 6 people instead of five times that amount.
The Main Course
In addition to the smoky grilled salad, Daniel served tarragon cornish hens with a cognac cream sauce loaded with chanterelles and grapes, and wild rice with grilled ramps (wild garlicky leeks). Dinner was served close to midnight, but it was a party so nobody cared.
We left that night (technically, early the next morning) with full bellies, new cooking tips, and nearly 5 hours of footage. I'm considering renaming the show "Cooking with Nada Surf".

FIGURE 11-10. Rules around all the elements reveal their element boxes.

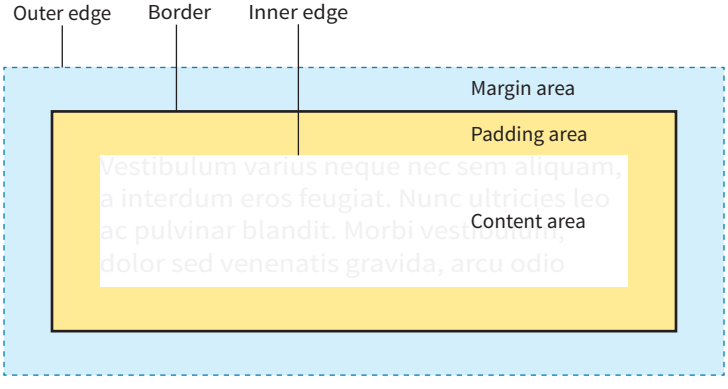


FIGURE 11-11. The CSS box model

CSS UNITS OF MEASUREMENT

This chapter lays the groundwork for upcoming lessons, so it's a good time to get familiar with the units of measurement used in CSS. Knowing how to use CSS units effectively is a core CSS skill. You'll be using them to set font sizes, the width and height of elements, margins, indents—basically any time a length value is appropriate.

The complete list is provided in the sidebar **"CSS Units."** Some will look familiar (like inches and millimeters), but there are some units that bear more explanation—I'll talk more about the absolute units and `rem`, `em`, and `vw`/`vh` in this section.

Absolute Units

Absolute units have predefined meanings or real-world equivalents. They are always the same size, regardless of the context in which they appear. The absolute units `pt`, `pc`, `in`, `mm`, `q`, and `cm` are irrelevant on screens, although they

CSS Units

CSS3 provides a variety of units of measurement. They fall into two broad categories: **absolute** and **relative**.

Absolute units

Absolute units have predefined meanings or real-world equivalents. With the exception of pixels, they are not appropriate for web pages that appear on screens.

`px`	Pixels (1 px = 1/96 inch in CSS)
`in`	Inches
`mm`	Millimeters
`cm`	Centimeters
`q`	¼ millimeter
`pt`	Points (1 pt = 1/72 inch); points are a unit commonly used in print design.
`pc`	Picas (1 pica = 12 points or 1/6 inch); picas are a unit commonly used in print design

Relative units

Relative units are based on the size of something else, such as the default text size or the size of the parent element. The container query units are not listed here; they will be covered in **Chapter 19, Responsive Web Design**.

`em`	A unit of measurement equal to the current font size
`ex`	x-height, approximately the height of a lowercase x in the font
`rem`	Root em, equal to the em size of the root element (`html`)
`ch`	Zero width, equal to the width of a zero (0) in the current font and size
`vw`	Viewport width unit, equal to 1/100 of the current viewport (browser window) width
`vh`	Viewport height unit, equal to 1/100 of the current viewport height
`vmin`	Viewport minimum unit, equal to the value of `vw` or `vh`, whichever is smaller
`vmax`	viewport maximum unit, equal to the value of `vw` or `vh`, whichever is larger

NOTES

- Although not a "unit," **percentages** are another common measurement value for web page elements. Percentages are calculated relative to another value, such as the value of a property applied to the current element or its parent or ancestor. The spec always says what a percentage value for a property is calculated on.
- Child elements do not inherit the relative values of their parent, but rather the resulting *calculated* values.
- Another type of relative unit, **container query length units**, are based on the size of the containing element. They are discussed in **Chapter 19, Responsive Web Design**.

may be useful for print style sheets. That narrows down your unit choices a bit.

The most popular absolute unit for web design is the pixel (**px**), which CSS defines as 1/96 inch. Pixels are right at home on a pixel-based screen. For a while, pixels were all we used; then we realized they are too rigid for pages that need to adapt to a wide variety of screen sizes and user preferences. Relative measurements like rem, em, and percentages are more appropriate to the fluid nature of the medium.

That said, pixels do still have their place in web design for elements that truly should stay the same size regardless of context. Border widths are appropriate in pixels, as are images that have inherent pixel dimensions.

Relative Units

As I just established, relative units are the way to go for most web measurements. The most useful options are **rem**, **em**, and **vw**/**vh** (see **Note**).

NOTE

There are also relative units based on the size of a container element (container query units), which I will introduce when we talk about container queries in **Chapter 19, Responsive Web Design**.

The rem unit

CSS Values and Units Module Level 3 introduced a relative measurement called a **rem** (for **root em**) that is based on the font size of the root (**html**) element, whatever size that happens to be. In modern browsers, the default root font size is 16 pixels; therefore, a rem is equivalent to a 16-pixel unit (unless you set it explicitly to another value). An element with its width set to 10rem would measure 160 pixels wide.

For the most part, you can use rem units like an absolute measurement in style rules; however, because it is relative, if the base font size changes, so does the size of a rem. If a user changes the base font size to 24 pixels for easier reading from a distance, for example, or if the page is displayed on a device that has a default font size of 24 pixels, the width of that 10rem element increases to 240 pixels. That may seem dodgy, but rest assured that it is a feature, not a bug. There are many instances in which you want a layout element to expand should the text size increase. It keeps the page proportional with the font size, which can help maintain optimum line lengths.

The em unit

An **em** is a relative unit of measurement that, in traditional typography, is based on the width of the capital letter *M* (thus the name "em"). In the CSS specification, an em is calculated as the distance between baselines when the font is set without any extra space between the lines (also known as **leading**). For text with a font size of 16 pixels, an em measures 16 pixels; for 12-pixel text, an em equals 12 pixels; and so on, as shown in FIGURE 11-12.

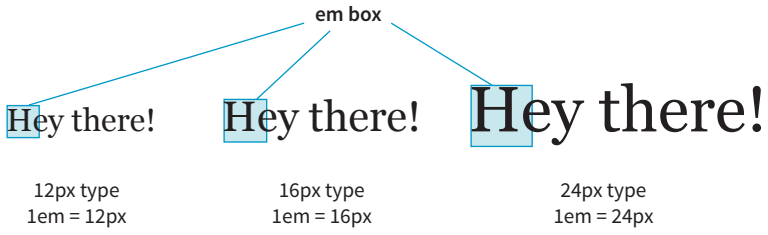


FIGURE 11-12. An em is based on the size of the text.

Once the dimension of an em for a text element is calculated by the browser, it can be used for all sorts of other measurements, such as indents, margins, the width of an element on the page, and so on. Basing measurements on text size helps keep everything in proportion should the text be resized.

The trick to working with ems is to remember they are always relative to the current font size of the element. To borrow an example from Eric Meyer and Estelle Weyl's *CSS: The Definitive Guide* (O'Reilly), if you set a 2em left margin on an **h1**, an **h2**, and a **p**, those elements will not line up nicely, because the em units are based on their respective elements' sizes (FIGURE 11-13).

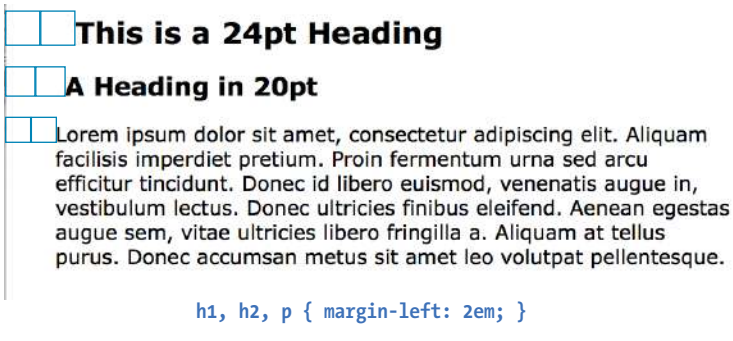

This is a 24pt Heading
A Heading in 20pt
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam facilisis imperdiet pretium. Proin fermentum urna sed arcu efficitur tincidunt. Donec id libero euismod, venenatis augue in, vestibulum lectus. Donec ultricies finibus eleifend. Aenean egestas augue sem, vitae ultricies libero fringilla a. Aliquam at tellus purus. Donec accumsan metus sit amet leo volutpat pellentesque.
h1, h2, p { margin-left: 2em; }

FIGURE 11-13. Em measurements are always relative to the element's font size. An em for one element may not be the same as for another.

NOTE

Don't confuse the em unit of measurement with the `em` *HTML element used to indicate emphasized text. They are totally different things.*

Viewport percentage lengths (vw/vh)

The viewport width (**vw**) and viewport height (**vh**) units are relative to the size of the viewport (browser window). One **vw** is equal to 1/100 the width of the viewport. Similarly, a **vh** is equal to 1/100 the height of the viewport. Viewport-based units are useful for making images and text elements stay the full width and/or height of the viewport:

```
header {
  width: 100vw;
  height: 100vh;
}
```

It's also easy to use **vw** and **vh** units to set the width and height of an element to a percentage of the display size—for example, to the equivalent of 50%:

```
img {
  width: 50vw;
  height: 50vh;
}
```

Related are the **vmin** unit (equal to the value of **vw** or **vh**, whichever is smaller) and **vmax** (equal to the value of **vw** or **vh**, whichever is larger). CSS Values and Units Module Level 4, still in Working Draft status, added 20 more viewport-relative values. For example, the **dvw** and **dvh** "dynamic" viewport units calculate viewport size without the browser and operating system **chrome** (the area of the screen reserved for the UI of the software itself), resulting in a truer measurement of the available rendering space on the screen. I'll leave the other viewport-relative units for you to explore on your own.

That should give you a good introduction to the units you'll be using in your style sheets. I recommend reading the full CSS module (***w3.org/TR/css-values-4/***) to deepen your knowledge and make the values listed for properties in this book easier to understand. In addition to length units, it includes text-based values (such as keywords, text strings, and URLs), numbers and percentage values, colors, and various units I don't have room to cover here (like that avalanche of new viewport units!). Be sure to check ***caniuse.com*** for values not listed in this book. They may be new and not fully supported.

DEVELOPER TOOLS RIGHT IN YOUR BROWSER

Because of the cascade, a single page element may have styles applied from a number of sources. This can make it tricky to debug a page when styles aren't displaying the way you think they should. Fortunately, every major browser comes with developer tools that can help you sort things out.

I've opened the simple *cooking.html* document that we've been working on in the Chrome browser, then selected View → Developer → Developer Tools from the menu. The DevTools panel is shown docked to the bottom of the document in FIGURE 11-14 (see **Note**).

In the Elements tab on the left, you can see the HTML source for the document. The content is initially hidden so you can see the structure of the document more clearly, but clicking the arrows opens each section. When you click an element in the source (like the second **p** element shown in the figure), that element is also highlighted in the browser window view.

The Styles tab on the right shows all of the styles that are being applied to the selected element. In the example, I can see the **font-size**, **font-family**, and **margin-left** properties coming from the embedded style sheet in the document. If there were external CSS documents, they'd be listed too. You can also

NOTE

You may find that the Chrome DevTools panel does not open at the bottom of the browser window. You can "dock" the tools panel to the left, right, or bottom of the window or display it in a separate window by selecting a "Dock Side" in the additional options. (The icon is the three stacked dots next to the Settings gear icon.)

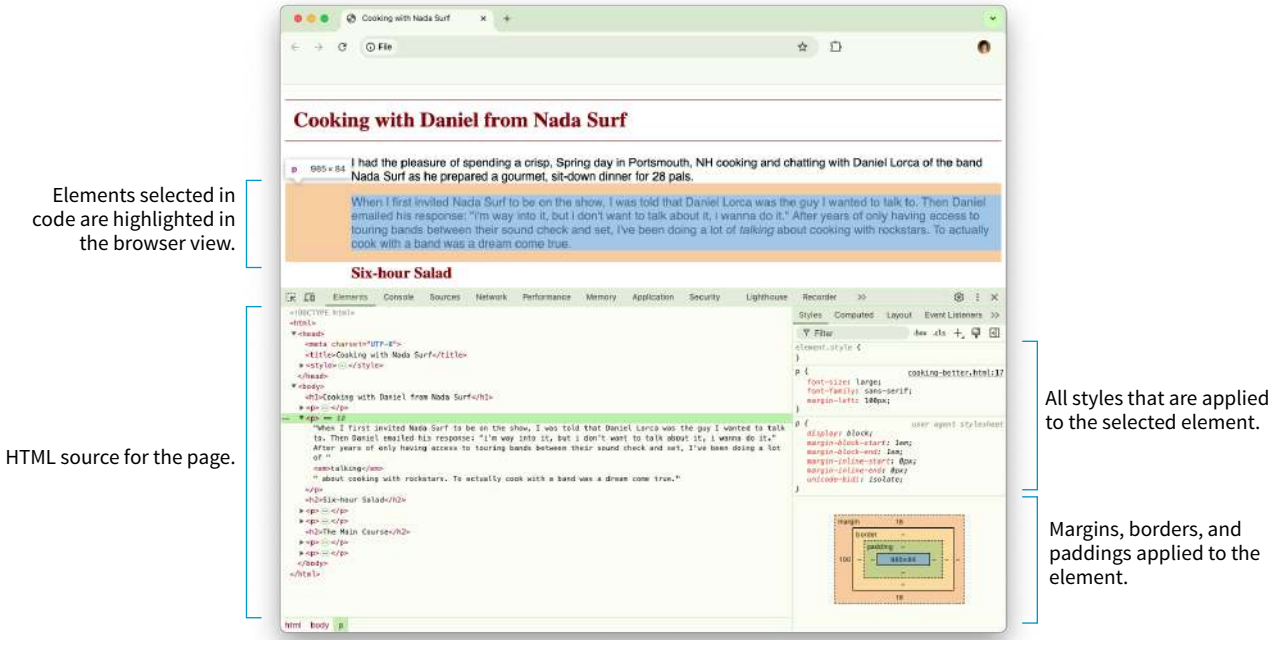

Elements selected in code are highlighted in the browser view.
HTML source for the page.
All styles that are applied to the selected element.
Margins, borders, and paddings applied to the element.
Cooking with Daniel from Nada Surf
I had the pleasure of spending a crisp, Spring day in Portsmouth, NH cooking and chatting with Daniel Lorca of the band Nada Surf as he prepared a gourmet, sit-down dinner for 28 pals.
When I first invited Nada Surf to be on the show, I was told that Daniel Lorca was the guy I wanted to talk to. Then Daniel emailed his response: "I'm way into it, but I don't want to talk about it, I wanna do it." After years of only having access to touring bands between their sound check and set, I've been doing a lot of talking about cooking with rockstars. To actually cook with a band was a dream come true.
Six-hour Salad

FIGURE 11-14. The Chrome browser with the DevTools panel open

see the "user agent stylesheet," which is the browser's default styles. In this case, the browser's style sheet adds the margin space around the paragraph. At the bottom of the Styles tab, Chrome provides a box model diagram for the selected element that shows the content dimensions, padding, border, and margins that are applied. This is a great tool for troubleshooting unexpected spacing in layouts.

The *super cool* thing is that when you edit the style rules in the panel, the changes are reflected in the browser view of the page in real time! If you select the `h1` element and change the color from maroon to green, it'll turn green in the window. This is a great way to experiment with or troubleshoot a design; however, it's just a preview, so if you want to make the changes in the document itself you'll have to duplicate them in your source.

You can inspect *any* page on the web in this way, play around with turning styles off and on, and even add some of your own. Nothing you do has any effect on the actual page, so it's just for your education and amusement.

The element and style inspectors are just the tip of the iceberg of what browser developer tools can do. You can also tweak and debug JavaScript, check performance, view the document in various device simulations, and much more. The good news is that all major browsers now have built-in tools with similar features. As a web developer, you'll find they are your best friend.

- Chrome DevTools (View → Developer → Developer Tools)
  developer.chrome.com/devtools

- Safari (Develop → Show Web Inspector)
  developer.apple.com/safari/tools/
- Firefox (Tools → Browser Tools → Web Developer)
  developer.mozilla.org/en-US/docs/Tools
- Microsoft Edge (open with F12 key)
  oreil.ly/wt3Mm
- Opera (Developer → Developer Tools)
  opera.com/browsers/opera/developer

MOVING FORWARD WITH CSS

This chapter covered the fundamentals of Cascading Style Sheets, including rule syntax, ways to apply styles to a document, and the central concepts of inheritance, the cascade (including priority, specificity, and rule order), and the box model. Style sheets should no longer be a mystery, and from this point on, we'll merely be building on this foundation by adding properties and selectors to your arsenal and expanding on the concepts introduced here.

CSS is a vast topic, well beyond the scope of this book. Bookstores and the web are loaded with information about CSS for all skill levels. I've compiled a list of the resources I've found the most useful during the learning process.

Books

There is no shortage of good books on CSS out there, but the only one that covers it all (it's *huge!*) is *CSS: The Definitive Guide*, 5th edition, by Eric A. Meyer and Estelle Weyl (O'Reilly).

Online Resources

The sites listed here are good starting points for online exploration of style sheets:

World Wide Web Consortium (*w3.org/TR/CSS/*)

The World Wide Web Consortium oversees the development of web technologies, including CSS. This page is a "snapshot" of the CSS specifications. See also *w3.org/Style/CSS/current-work*.

MDN Web Docs (*developer.mozilla.org*)

The CSS pages at MDN include detailed reference pages, step-by-step tutorials, and demos. It's a great hub for researching any web technology.

CSS-Tricks (*css-tricks.com*)

This site is my go-to for articles on all topics related to CSS. You'll find a lot of references to it throughout this book.

Smashing Magazine (*smashingmagazine.com/category/css/*)

This online magazine features some of the best thinking and writing on cutting-edge, standards-based web design, including CSS.

Inspiration

If you are looking for design inspiration, these sites (as well as CSS Zen Garden, mentioned earlier) are fun to poke around in. A web search for "web design gallery" or "web design inspiration" will turn up plenty more:

CSS Nectar (*cssnectar.com*)

A curated gallery of exceptional site design from around the world.

Squarespace templates (*squarespace.com/templates*)

Squarespace is a popular site hosting service. It offers a wide range of predesigned templates that are fun to browse through.

CSS Design Awards (*cssdesignawards.com*)

Another gallery showcasing exceptional responsive design websites.

TEST YOURSELF

Here are a few questions to test your knowledge of the CSS basics. Answers are provided in **Appendix A**.

1. Identify the various parts of this style rule:

```
blockquote { line-height: 1.5; }
```

selector: _____ value: _____

property: _____ declaration: _____

2. What color will paragraphs be when this embedded style sheet is applied to a document? Why?

```
<style>
   p { color: purple; }
   p { color: green; }
   p { color: gray; }
</style>
```

3. Rewrite each of these CSS examples. Some of them are completely incorrect, and some could just be written more efficiently.

   a.
   ```
   p {font-family: sans-serif;}
   p {font-size: 1em;}
   p {line-height: 1.2em;}
   ```

   b.
   ```
   blockquote {
   font-size: 1em
   line-height: 150%
   color: gray }
   ```

   c.
   ```
   body
   {background-color: black;}
   {color: #666;}
   {margin-left: 12em;}
   {margin-right: 12em;}
   ```

   d.
   ```
   p {color: white;}
   blockquote {color: white;}
   li {color: white;}
   ```

   e.
   ```
   <strong style="red">Act now!</strong>
   ```

4. Circle all the elements that you would expect to appear in red when the following style rule is applied to a document with the structure diagrammed in FIGURE 11-15:

   ```
   div#intro { color: red;}
   ```

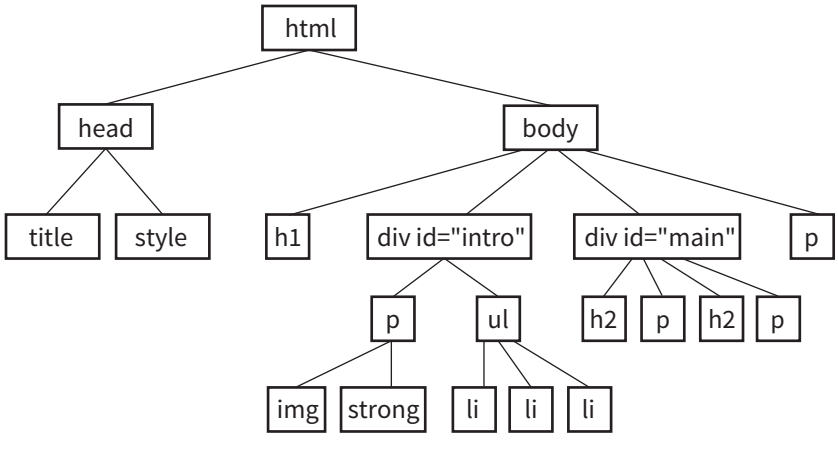


FIGURE 11-15. The document structure of a sample document

CHAPTER 12

FORMATTING TEXT

Now that you've gotten your feet wet formatting text, are you ready to jump into the deep end? By the end of this chapter, you'll have picked up over 30 additional CSS properties used to manipulate the appearance of text. Along the way, you'll learn how to use more powerful selectors for targeting elements in a particular context and with a specific `id` or `class` name. We'll also look at a way to calculate selector specificity.

The nature of the web makes specifying type tricky, if not downright frustrating, particularly if you have experience designing for print or even formatting text in a word processing program. There is no way to know for sure whether the font you specify will be available or how large or small the type will appear when it hits your users' browsers. I'll address the best practices for dealing with these challenges as we go along.

Throughout this chapter, we'll be sprucing up a Black Goose Bistro online menu similar to the one we marked up back in **Chapter 5, Marking Up Text**. I encourage you to work along with the exercises to get a feel for how the properties work. FIGURE 12-1 shows how the menu looks initially, and how it will look when we're done. It's not a masterpiece, because we're just scratching the surface of CSS here, but at least the text has a little more personality.

IN THIS CHAPTER

Font properties

Web fonts

Text line settings

Text effects

Selectors: descendant, ID, and class

Specificity overview

List styles

NOTE

As of this writing, the most recently published standard for fonts is CSS Fonts Module Level 3 (w3.org/TR/css-fonts-3/). It is also the best supported in modern browsers and is the reference version for the properties in this chapter. CSS Fonts Module Level 4 is in Working Draft status (w3.org/TR/css-fonts-4/). Although many of the features are still experimental, it's fun to look at the spec to see where web typography is headed.

BASIC FONT PROPERTIES

When designing a text document (for print or the web), it is a common practice to specify the font first. In CSS, a set of font-related properties for typeface, size, weight, font style, and special characters handle typography design. There are also shortcut properties that let you specify multiple font attributes in a single rule.

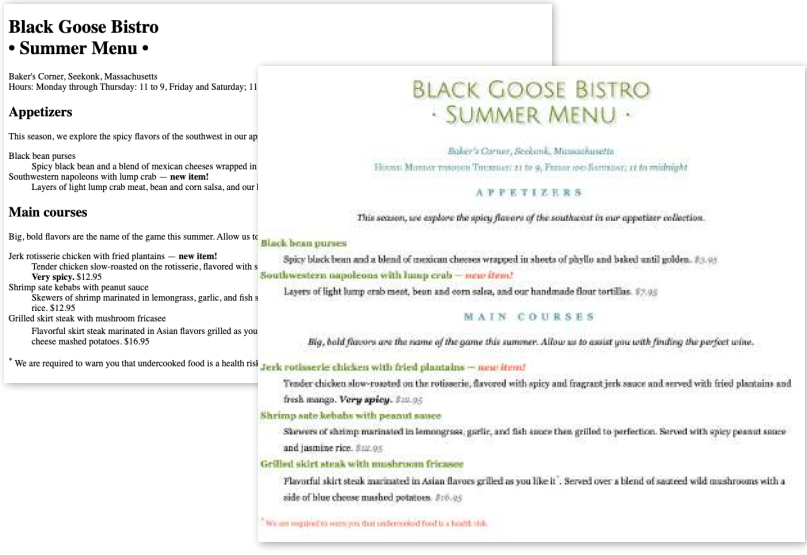

Black Goose Bistro
• Summer Menu •
Appetizers
Main courses
BLACK GOOSE BISTRO
· SUMMER MENU ·
APPETIZERS
MAIN COURSES
Black bean purses
Southwestern napoleons with lump crab — new item!
Jerk rotisserie chicken with fried plantains — new item!
Shrimp sate kebabs with peanut sauce
Grilled skirt steak with mushroom fricasee

FIGURE 12-1. Before and after views of the Black Goose Bistro menu that we'll be working on in this chapter

A Word About Property Listings

Each CSS property listing in this book is accompanied by information on how it behaves and how to use it. Property listings include:

Values:
These are the accepted values for the property. Predefined keyword values appear in code font (for example, `small`, `italic`, or `small-caps`) and must be typed in exactly as shown.

Default:
This is the value that will be used for the property by default (its **initial value**)—that is, if no other value is specified. Note that the default browser style sheet values may vary from the defaults defined in CSS.

Applies to:
Some properties apply only to certain types of elements.

Inherits:
This indicates whether the property is passed down to the element's descendants.

CSS-wide keywords

All CSS properties accept the four CSS-wide keywords `initial`, `inherit`, `unset`, and `revert`. Because they are shared by all properties, they are not listed with the values for individual property listings:

- The `initial` keyword explicitly sets the property to its default (initial) value.
- The `inherit` keyword allows you to explicitly force an element to inherit a style property from its parent. This may come in handy to override other styles applied to that element and to guarantee that the element always matches its parent.
- The `unset` keyword erases declared values occurring earlier in the cascade, setting the property to either `inherit` or `initial`, depending on whether it inherits or not.
- The `revert` keyword (added in the CSS Cascading and Inheritance Level 4 module) reverts the property value to whatever value it would have had if no changes had been made in the current style sheet. For example, if the author of the document sets a property to `revert`, the user styles will be used if present, and if not, it goes back to the user agent (browser default) styles.

For a more detailed explanation, see the MDN Web Docs description (*oreil.ly/f-TEC*), which is a little easier to read than the spec (*oreil.ly/QP2bS*).

Specifying the Font Name

Choosing a typeface, or **font family** as it is called in CSS, for your text is a good place to start. Let's begin with the `font-family` property and its values.

font-family

Values: *One or more font or generic font family names, separated by commas*

Default: Depends on the browser

Applies to: All elements

Inherits: Yes

Use the `font-family` property to specify a font or list of fonts (known as a **font stack**) by name, as shown in these examples:

```
body { font-family: Arial; }
var { font-family: Courier, monospace; }
p { font-family: "Duru Sans", Verdana, sans-serif; }
```

Here are some important syntax requirements:

- Use commas to separate multiple font names, as shown in the second and third examples.
- Notice that font names that contain a character space (such as Duru Sans in the third example) must appear within quotation marks.
- It is a strong convention to capitalize names of fonts (for example, `Arial` instead of `arial`), but it is not required according to the spec. Generic font families (for example, `sans-serif`) are always lowercase.

You might be asking, "Why specify more than one font?" That's a good question, and it brings us to one of the challenges of specifying fonts for the web.

Font limitations

Browsers are limited to displaying fonts they have access to. In the past, that meant the fonts that were already installed on the user's hard drive. In 2010, however, there was a boom in browser support for embedded web fonts using the CSS `@font-face` rule, so it became possible for designers to provide their own fonts, as I'll discuss soon.

But back to our `font-family` property. Even when you specify that the font should be Futura in a style rule, if the browser can't find it (for example, if that font is not installed on the user's computer or the provided web font fails to load), it uses its default font instead.

Fortunately, CSS allows us to provide a list of backup fonts (that font stack you saw earlier) should our first choice not be available. If the first specified font is not found, the browser tries the next one, and so on down the list until it finds one that works. In the third `font-family` rule shown in the previous code example, if the browser does not find Duru Sans, it will use Verdana, and if Verdana is not available, it will substitute some other sans-serif font.

■ AT A GLANCE

Font Properties

The CSS2.1 font-related properties are universally supported:

```
font-family
font-size
font-weight
font-style
font-variant
font
```

CSS Fonts Module Level 3 added these properties for more sophisticated font handling:

```
font-stretch
font-variant-ligatures
font-variant-position
font-variant-caps
font-variant-numeric
font-variant-alternates
font-variant-east-asian
font-size-adjust
font-kerning
font-feature-settings
font-language-override
```

■ TERMINOLOGY

Typeface Versus Font

A **typeface** is a particular design of a set of characters. For example, Helvetica and Georgia are both typefaces. A **font** is a particular variation of a typeface. Helvetica Bold, Helvetica Italic, and Helvetica Condensed are all individual fonts that are part of the Helvetica typeface.

Generic font families

That last option, "some other sans-serif font," bears more discussion. "Sans-serif" is one of the generic font families that you can specify with the `font-family` property. When you specify a generic font family, the browser chooses an available font from that stylistic category. FIGURE 12-2 shows examples from each family.

FIGURE 12-2. Examples of the universally supported generic font families

The universally supported font family values are:

`serif`

Examples: Times, Times New Roman, Georgia, Cambria, Garamond

Serif typefaces have decorative slab-like appendages (serifs) on the ends of certain letter strokes.

`sans-serif`

Examples: Arial, Arial Black, Verdana, Trebuchet MS, Helvetica, Geneva, Century Gothic

Sans-serif typefaces have straight letter strokes that do not end in serifs.

`monospace`

Examples: Courier, Courier New, Monaco, Andale Mono

In monospace (also called constant width) typefaces, all characters take up the same amount of space on a line. For example, a capital *W* will be no wider than a lowercase *i*. Compare this to proportional typefaces (such as the one you're reading now) that allot different widths to different characters.

`cursive`

Examples: Apple Chancery, Zapf-Chancery, Lucinda, Comic Sans

Cursive fonts emulate a script or handwritten appearance.

`fantasy`

Examples: Impact, Western, or other decorative font

Fantasy fonts are purely decorative and may be appropriate for headlines and other display type.

Web fonts

If you aren't satisfied designing with generic font families and fonts commonly installed with operating systems, you can specify the web font of your choice with the `@font-face` rule. It took a while to get consensus, but virtually all modern browsers now support the **WOFF2** (for **Web Open Font Format** version 2) font file format. WOFF is a container that packages fonts for web delivery (see **Note**). WOFF2 supports TrueType and OpenType features, including variable fonts and font collections.

There are two methods for embedding web fonts on your site. One option is to upload the appropriate font files to your own server and host them yourself. The other is to link to fonts hosted by a font embedding service. Let's take a brief look at each approach.

Self-hosting

You may choose to purchase fonts and host them on your own server. The advantage is you have more control, and it may be better for security as you do not open your site to a third-party service that may share data. Of course, that doesn't mean it's risk-free (see **Warning**).

Keep in mind that most typefaces aren't free and come with licenses that cover very specific uses by a limited number of machines. If you purchase a font, be sure to check the End User License Agreement (EULA) to see if web usage is included. However, thanks to demand, foundries are opening more fonts up

New Generic Font Families

CSS Fonts Module Level 4 includes additional generic font families, including some for languages with non-Western characters. As of this writing, they are still poorly supported, but they may be viable in the future. The additions are:

- `system-ui`, `ui-serif`, `ui-sans-serif`, `ui-monospace`, and `ui-rounded`. These specify the font used by the user's operating system. Their purpose is to "allow web content to integrate with the look and feel of the native OS."
- `generic(fangsong)` and `generic(kai)` for Chinese typefaces.
- `generic(nastaliq)` for Urdu and Kashmiri.
- `math`, a font family for mathematical expressions.

NOTE

If you need to support very old browsers that do not support WOFF2 (such as Internet Explorer), you can provide a list of alternate font file formats. For a good overview of web fonts, including how to generate alternate versions and the code to support them, see the MDN Web Docs article "Web Fonts" (oreil.ly/kJ_-n).

WARNING

As with any file you download from the internet to your computer, there is the risk that a font file could bring in malware or viruses. Be sure to purchase and download fonts only from trustworthy foundries and services.

for web use, and there are a growing number of open source fonts that you can use for free. Ethan Dunham's Fontspring (*fontspring.com*) is a good place to purchase fonts that have a web license. Font Squirrel (*fontsquirrel.com*), also by Ethan Dunham, is a great source for open source fonts that can be used for commercial purposes for free (FIGURE 12-3).

It's worth noting that for standard fonts, each font file corresponds to a single weight or variant of a typeface, so if you want to use regular, bold, and italic versions, you have to host and declare three different font files. We now also have access to variable fonts that contain all variations of a font in one file. See the **"Variable Fonts"** sidebar for an introduction.

Cloud font embedding services

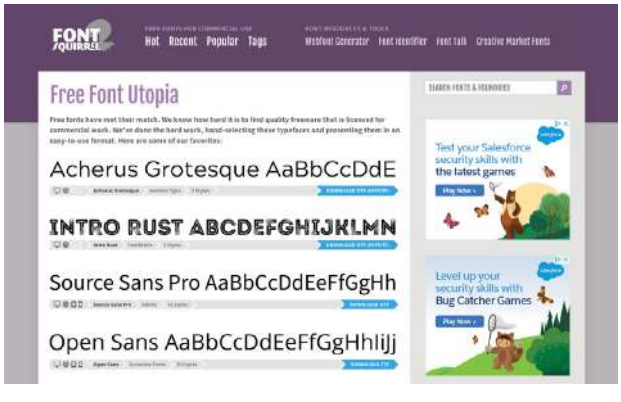
If hosting your own fonts feels like a lot of responsibility, you may want to sign up with one of the font embedding services. You get access to high-quality fonts, and the service handles font licensing and font protection for the foundries. They also generally provide an interface and tools that make embedding a font as easy as copy and paste.

Google Fonts (*fonts.google.com*) is a great place to start, because all of the fonts are open source and free to use. If you have a budget, you can pay a fee to use Adobe Fonts (*fonts.adobe.com*), Hoefler & Co. (*typography.com/webfonts*), or one of the many other services you'll find online. They have a variety of fee structures, including monthly fees and charging by the font. Some have a surcharge for bandwidth as well, so be sure to check the terms.

The disadvantages of using third-party font hosting services include a potential hit to performance and rendering speed (the page may draw a few times while the font loads), and privacy concerns due to font requests having the potential to be tracked and the data stored.

FONTSPRING
"Amo il bello ed il buono ovunque si trovino"
Pellegrino Artusi
Trung tâm
Maître des
Hinn ágæti leikari
PAD→CNM
1 x Open Return
This train consists of 8 carriages, first class is located at the front.

FontSpring (*fontspring.com*)



Font Squirrel (*fontsquirrel.com*)

FIGURE 12-3. Examples of exciting fonts available for use on web pages at Fontspring (left) and Font Squirrel (right). There are many marketplaces for fonts, but be sure to check that they are licensed for commercial and web use.

Variable Fonts

Variable fonts take advantage of a feature in OpenType that allows multiple variations of a typeface to be delivered in one file (unlike standard fonts that require a separate file for each variation). The cool thing is, you don't just get access to the standard regular and bold versions; you can set the precise amount of boldness you want on a sliding scale from 1 to 1,000. Weight is just one **variation axis** that is standardized for variable fonts. The other registered axes are width, slant, italic, and optical size. FIGURE 12-4 shows just a few combinations of width and weight for the Helvetica Now Variable font.

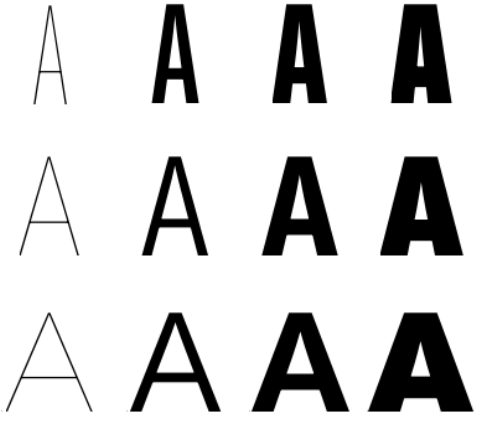


FIGURE 12-4. Examples of different width and weight settings for a variable font

Font designers may also choose to add custom variation axes to their fonts, allowing adjustments to all kinds of things, like the height of ascenders and descenders, how "puffy" or "pointy" the font appears, and the appearance of numbers, just to name a few.

The advantage of variable fonts is having access to the full range of styles for a typeface, so you can customize it to suit its use perfectly. A variable font generally has a smaller file size and requires fewer requests to the server than a pile of individual font files, so there are performance benefits as well.

A deep exploration of variable fonts is beyond the scope of this book, but here are some resources to get you started:

- The best place to dive into variable fonts is the "Variable Fonts Primer" at *variablefonts.io*.
- MDN Web Docs provides an overview at *oreil.ly/1p6OF*.
- Google Fonts has an introduction at *oreil.ly/K7KrU*.
- The Variable Fonts site maintained by Nick Herman at *v-fonts.com* allows you to play with a large collection of variable fonts that are available for download. You can find free fonts on the Licenses page.

The @font-face rule

The @font-face "at-rule" gives the font a name that you can then reference later with font-family.

Once you've identified the font you want to use, you need to specify it in your style sheet. The **`@font-face`** "at-rule" gives the font a name that you can then reference later with **`font-family`**. Although you can name it anything you like, it makes sense to give it a name that represents the actual name of the font. The **`@font-face`** rule uses the **`src`** property to provide the font's URL and file format.

This example establishes the name "MyWebFont" for a *webfont.woff2* font that we're hosting on our own hypothetical server. Once the web font is named and located, we can apply it to content (such as the **`p`** element in the example) with the **`font-family`** property, as shown earlier:

```
<style>
  @font-face {
    font-family: "MyWebFont";
    src: url(/fonts/webfont.woff2) format("woff2");
    font-weight: normal;
    font-style: normal;
  }
  ...
  p {
    font-family: MyWebFont, Georgia, serif;
  }
  ...
</style>
```

For variable fonts, the **`@font-face`** rule may include acceptable ranges for standard axes such as width, weight, and so on:

```
@font-face {
  font-family: "MyWebFont";
  src: url(/fonts/webfont.woff2) format("woff2-variations");
  font-weight: 100 1000;
  font-width: 0 50;
}
```

There's a lot more to say about the **`@font-face`** settings for variable fonts than I have room for here, so be sure to look at the resources in the **"Variable Fonts"** sidebar when you're ready to use one.

The good news is that if you are using a font embedding service, they will provide the **`@font-face`** code for you to ensure the URL to the font on their server is correct. You'll get a chance embed a Google Font in EXERCISE 12-1.

Preloading Fonts

If your page starts rendering before your web font has downloaded, your visitors may see a blank area (or FOIT, for Flash of Invisible Text) until your font arrives. To avoid that situation and improve the perceived performance of your site, it is possible to preload a font file so it's available when the page draws. Preloading causes the browser to give the font a higher priority in its download queue.

To preload a font file, use the **link** element:

```
<link rel="preload" href="/fonts/webfont.woff2" as="font"
type="font/woff2" crossorigin>
```

The **as** attribute identifies the resource as a font file (**as** can be used with other media types too, such as images, scripts, and styles). The **type** attribute is the MIME type of the font file. Finally, **crossorigin** is required for the preload to work; it has to do with media requests coming from external domains.

If your page uses multiple web fonts, it is recommended that you only preload the one that needs to appear on the first screen of the page and let the rest arrive as the browser naturally processes them.

Font stack strategies

The best practice for specifying fonts for web pages is to start with your first choice, provide some similar alternatives, and then end with a generic font family that at least gets users in the right stylistic ballpark. For example, if you want an upright, sans-serif font, you might start with a web font that you link to (Oswald), list a few that are commonly installed on users' machines (Univers, Tahoma, Geneva), and finish with the generic **sans-serif**:

```
font-family: Oswald, Univers, Tahoma, Geneva, sans-serif;
```

There is no limit to the number of fonts you can include, but many designers strive to keep it under 10. A good font stack should include stylistically related fonts that are known to be installed on most computers. Sticking with fonts that come with the Windows, macOS, and Linux operating systems, as well as fonts that get installed with popular software packages such as Microsoft Office and Adobe Creative Suite, gives you a solid list of "web-safe" fonts to choose from. A good place to look for stylistically related web-safe fonts is CSS Fonts, from Dan's Tools (***cssfontstack.com***).

As you can see, specifying fonts for the web is more like *suggesting* them. You don't have absolute control over which font your users will see. You might get your first choice; you might get the generic fallback. It's one of those web design quirks you learn to live with.

Now seems like a good time to get started formatting the Black Goose Bistro menu. We'll add new style rules one at a time as we learn new properties, starting with EXERCISE 12-1.

EXERCISE 12-1. Formatting a menu

In this exercise, we'll change the fonts for the body and main heading of the Black Goose Bistro menu document, *menu.html*, which is available at *learningwebdesign.com/6e/materials*. Open the document in a text or code editor. You can also open it in a browser to see its "before" state. It should look something like the image on the left in FIGURE 12-1. Hang on to this document, because this exercise will continue as you pick up additional font properties.

I've included an embedded font in this exercise to show you how easy it is to add one with a service like Google Fonts.

1. Use an embedded style sheet for this exercise. Start by adding a **style** element in the **head** of the document, like this:

```
<head>
 <title>Black Goose Bistro</title>
  <style>
  </style>
</head>
```

2. I would like the main text to appear in Verdana or some other sans-serif font. Instead of writing a rule for every element in the document, I can write one rule for the **body** element that will be inherited by all the elements it contains. Add this rule to the embedded style sheet:

```
<style>
  body {font-family: Verdana, sans-serif;}
</style>
```

3. Now let's try embedding a font. I want a fancy font for the "Black Goose Bistro • Summer Menu" headline, so I'll use a free display font called Julius Sans One from Google Fonts (*fonts.google.com*). Google provides the code for linking the font file on its server (it's actually a link to an external style sheet). The code must be placed in the **head** of the document, so copy it exactly as it appears, but *keep each* **link** *element on one line*. You could also copy it from the finished exercise file provided with this chapter's materials if that's easier. Put them before the **style** element:

```
<head>
  <title>Black Goose Bistro Summer Menu</title>
  <link rel="preconnect" href="https://fonts.→
googleapis.com">
  <link rel="preconnect" href="https://fonts.→
gstatic.com" crossorigin>
  <link href="https://fonts.googleapis.com/→
css2?family=Julius+Sans+One&display=swap"→
rel="stylesheet"><style>
  <style>
    ...
```

4. Now write a rule that applies the font to the **h1** element. Note that the font name is in quotation marks because it's three words. Specify Verdana and a sans-serif font as fallbacks in the font stack:

```
<style>
  body {font-family: Verdana, sans-serif;}
  h1 {font-family: "Julius Sans One", Verdana,
sans-serif;}
</style>
```

5. Save the document and reload the page in the browser. It should look like FIGURE 12-5. Note that you'll need to have an internet connection and a current browser to view the Julius Sans One headline font. We'll work on the text size in the next exercise.

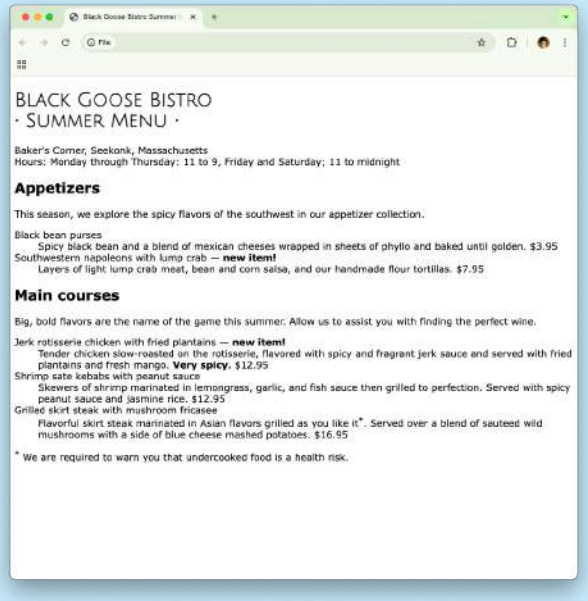

BLACK GOOSE BISTRO
· SUMMER MENU ·
Baker's Corner, Seekonk, Massachusetts
Hours: Monday through Thursday: 11 to 9, Friday and Saturday; 11 to midnight
Appetizers
This season, we explore the spicy flavors of the southwest in our appetizer collection.
Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden. $3.95
Southwestern napoleons with lump crab — new item!
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. $7.95
Main courses
Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.
Jerk rotisserie chicken with fried plantains — new item!
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. Very spicy. $12.95
Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice. $12.95
Grilled skirt steak with mushroom fricasee
Flavorful skirt steak marinated in Asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. $16.95
* We are required to warn you that undercooked food is a health risk.

FIGURE 12-5. The menu after we change only font families

Specifying Font Size

Use the aptly named **font-size** property to specify the size of the text.

font-size

Values: *length unit* | *percentage* | xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger

Default: medium

Applies to: All elements

Inherits: Yes

You can specify text size in several ways:

- Using one of the CSS length units, as shown here:

  ```
  h1 { font-size: 1.5em; }
  ```

  When specifying a number of units, be sure the unit abbreviation immediately follows the number, with no extra character space in between (see the sidebar **"Providing Measurement Values"**).

- As a percentage value, sized up or down from the element's inherited font size:

  ```
  h1 { font-size: 150%; }
  ```

- Using one of the absolute keywords **xx-small**, **x-small**, **small**, **medium**, **large**, **x-large**, or **xx-large** (on most current browsers, **medium** corresponds to the default font size):

  ```
  h1 { font-size: x-large; }
  ```

- Using a relative keyword (**larger** or **smaller**) to nudge the text larger or smaller than the surrounding text:

  ```
  strong { font-size: larger; }
  ```

To cut to the chase, despite all these options, the preferred values for **font-size** in contemporary web design are the relative length units **rem** and **em**, as well as percentage values. You can specify font size in pixels (**px**), but in general, this is problematic for accessibility. All of the other absolute units (**pt**, **pc**, **in**, etc.) are out, unless you are creating a style sheet specifically for print.

I'll explain the keyword-based **font-size** values in a moment, but let's start our discussion with the best practice of using relative values.

Sizing text with relative values

The best practice for setting the font size of web page elements is to do it in a way that respects the user's preferences. Relative sizing values (**%**, **rem**, and **em**) allow you to use the default font size as the basis for proportional sizing of other text elements. It's usually not important that the headlines are exactly 24 pixels; it *is* important that they are 1.5 times larger than the main text so they stand out. If the user changes their browser preferences to make

Providing Measurement Values

When you're providing measurement values, the unit must immediately follow the number, like this:

```
margin: 2em;
```

Adding a space before the unit, like this, will cause the property not to work:

```
margin: 2 em; /* wrong */
```

It is acceptable to omit the unit of measurement for zero values:

```
margin: 0;
```

CSS length units are discussed in **Chapter 11, Introducing Cascading Style Sheets**. See also the **"CSS Units Cheat Sheet"** sidebar.

■ AT A GLANCE

CSS Units Cheat Sheet

As a quick reference, here are the CSS length units again:

Relative units

em	ex	rem	ch
vw	vh	vmin	vmax

Absolute units

px	in	mm	cm
q	pt	pc	

the default font size larger, for example to compensate for poor eyesight, the headlines will appear larger, too.

To maintain the browser's default size, set the **font-size** of the root element to 100% (see **Note**):

```
html {
  font-size: 100%;
}
```

That sets the basis for relative sizing. Because the default font size for all modern browsers is 16 pixels, we'll assume our base size is 16 pixels going forward (but do keep in mind that it could be different).

NOTE

It is also common practice to set the **body** *to 100%, but setting the* **font-size** *property on the* **html** *element is preferable because it explicitly sets the root font size, which then will be used as the base for* **rem** *calculation.*

The preferred font-size values are em, rem, and %.

Rem values

The rem unit, which stands for "root em," is always relative to the size of the root (**html**) element. If the root size is 16 pixels, then 1 rem equals 16 pixels. What's nice about rem units is that because they are always relative to the same element, they are the same size wherever you use them throughout the document. In that way, they work like an absolute unit. However, should the root size be set to something other than 16 pixels, elements specified in rem values will resize accordingly and proportionally. It's the best of both worlds.

Here is an **h1** heading sized with rem values that has inherited the default 16-pixel font size from the root:

```
h1 { font-size: 1.5rem; }  /* 1.5 x 16 = 24 */
```

Em measurements

NOTE

It is possible to make text resize automatically based on the size of the viewport or a containing element in a technique called "fluid typography." Learn more about it in **Chapter 19, Responsive Web Design**.

Em units are based on the font size of the current element. When you specify **font-size** in ems, it will be relative to the inherited size for that element. Once the em size is calculated for an element, it can be used for other measurements as well, such as margins, padding, widths, and any other settings you want to always be relative to the size of the font.

Here, I've used em units to specify the size of an **h1** (in this example, we'll assume it has inherited the default 16-pixel font size):

```
h1 { font-size: 1.5em; }  /* 1.5 x 16 = 24 */
```

There are a few snags to working with ems. One is that because of rounding errors, there is some inconsistency in how browsers and platforms render text set in ems.

The other tricky aspect to using ems is that they are based on the *inherited* size of the element, which means that their size is based on the context in which they are applied.

The **h1** in the previous example was based on an inherited size of 16 pixels. But if this **h1** had appeared in an **article** element that had its font size set

to 14 pixels, it would inherit the 14-pixel size, and its resulting size would be just 21 pixels (1.5 × 14 = 21). Here's an example that demonstrates this; FIGURE 12-6 shows the results.

THE MARKUP

```
<h1>Headline in Body</h1>
<p>Pellentesque ligula leo...</p>
<article>
  <h1>Headline in Article</h1>
  <p>Vivamus...</p>
</article>
```

THE STYLES

```
h1 {
  font-size: 1.5em;    /* sets all h1s to 1.5em */
}
article {
  font-size: .875em    /* 14 pixels based on 16px default */
}
```

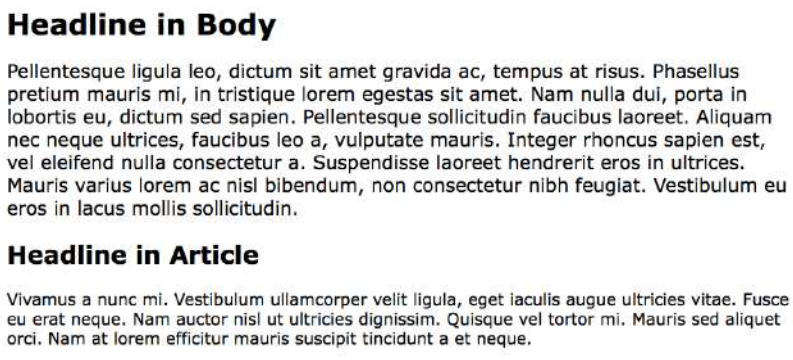

Headline in Body
Pellentesque ligula leo, dictum sit amet gravida ac, tempus at risus. Phasellus pretium mauris mi, in tristique lorem egestas sit amet. Nam nulla dui, porta in lobortis eu, dictum sed sapien. Pellentesque sollicitudin faucibus laoreet. Aliquam nec neque ultrices, faucibus leo a, vulputate mauris. Integer rhoncus sapien est, vel eleifend nulla consectetur a. Suspendisse laoreet hendrerit eros in ultrices. Mauris varius lorem ac nisl bibendum, non consectetur nibh feugiat. Vestibulum eu eros in lacus mollis sollicitudin.
Headline in Article
Vivamus a nunc mi. Vestibulum ullamcorper velit ligula, eget iaculis augue ultricies vitae. Fusce eu erat neque. Nam auctor nisl ut ultricies dignissim. Quisque vel tortor mi. Mauris sed aliquet orci. Nam at lorem efficitur mauris suscipit tincidunt a et neque.

FIGURE 12-6. All `h1` elements are sized at 1.5em, but they are different sizes because of the context in which they appear.

From this example, you can see that an element set in ems might appear at different sizes in different parts of the document. If you wanted the **`h1`** in the **`article`** to be 24 pixels as well, you could calculate the em value by dividing the target size by its context: 24 / 14 = 1.71428571em. (No need to round that figure down...the browser knows what to do with it.)

If you have elements nested several layers deep, the size increase or decrease compounds, which can create problems. With many layers of nesting, text may end up being way too small. When working with ems, pay close attention and write style rules in a way that takes the context into account.

This compounding nature of the em is what has driven the popularity of the predictable rem unit.

To calculate % and em values, use this formula: target size ÷ size of context = result.

Percentage values

We saw a percentage value (100%) used to preserve the default font size in an earlier example, but you can use percentage values for any element. They are pretty straightforward.

In this example, the `h1` inherits the default 16px size from the `html` element, and applying the value 150% multiplies that *inherited* value, resulting in an `h1` sized at 24 pixels:

```
h1 { font-size: 150%; }  /* 150% of 16 = 24 */
```

Working with keywords

An alternative way to specify `font-size` is by using one of the predefined absolute keywords: `xx-small`, `x-small`, `small`, `medium`, `large`, `x-large`, and `xx-large`. The keywords do not correspond to particular measurements, but rather are scaled consistently in relation to one another. The default size is `medium` in current browsers. FIGURE 12-7 shows how each of the absolute keywords renders in a browser when the default text is set at 16 pixels. I've included samples in Verdana and Times to show that, even with the same base size, there is a big difference in legibility at sizes `small` and below. Verdana was designed to be legible on screens at small font sizes; Times was designed for print so is less legible in that context.

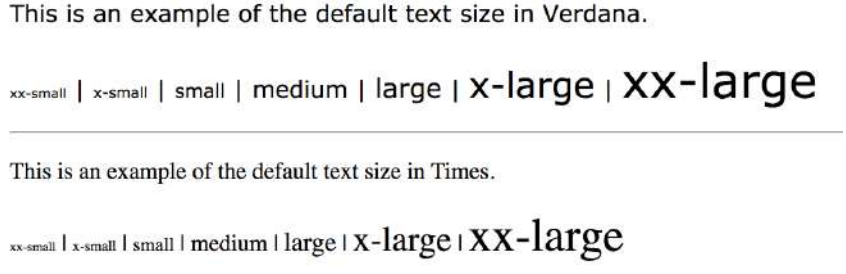

This is an example of the default text size in Verdana.
xx-small | x-small | small | medium | large | x-large | xx-large
This is an example of the default text size in Times.
xx-small | x-small | small | medium | large | x-large | xx-large

FIGURE 12-7. Text sized with absolute keywords

The relative keywords `larger` and `smaller` are used to shift the size of text relative to the size of the parent element's text. The exact amount of the size change is determined by each browser and is out of your control. Despite that limitation, this is an easy way to nudge type a bit larger or smaller if the exact proportions are not critical.

You can apply your new CSS font-sizing knowledge in EXERCISE 12-2.

EXERCISE 12-2. Setting font size

Let's refine the size of some of the text elements to give the online menu a more sophisticated appearance. You can save the document at any point and take a peek in your browser to see the results of your work. You should also feel free to try out other size values along the way.

1. There are many approaches to sizing text on web pages. In this example, start by putting a stake in the ground and setting the `font-size` of the `html` element to 100%, thus clearing the way for rem measurements thereafter. Add this new rule to the style sheet:

```
html {
  font-size: 100%;
}
```

2. The browser default of 16 pixels is a fine size for the main page text, but I would like to improve the appearance of the heading levels. Let's make the main heading 40 pixels, or two and a half times larger than the body text. Add a new rule that sets the size of the `h1` to 2.5rem (note that we could use 250% to achieve the same thing):

```
h1 {
  font-family: "Julius Sans One", Verdana, sans-serif;
  font-size: 2.5rem;
}
```

3. Now make the `h2`s the same size as the body text so they don't stand out so much:

```
h2 {
  font-size: 1rem;
}
```

FIGURE 12-8 shows the result of our font-sizing efforts.

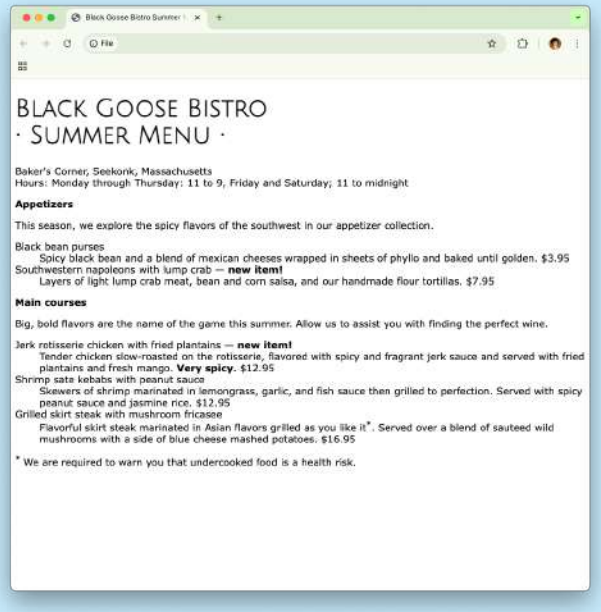

BLACK GOOSE BISTRO
· SUMMER MENU ·
Baker's Corner, Seekonk, Massachusetts
Hours: Monday through Thursday: 11 to 9, Friday and Saturday: 11 to midnight
Appetizers
This season, we explore the spicy flavors of the southwest in our appetizer collection.
Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden. $3.95
Southwestern napoleons with lump crab — new item!
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. $7.95
Main courses
Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.
Jerk rotisserie chicken with fried plantains — new item!
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. Very spicy. $12.95
Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice. $12.95
Grilled skirt steak with mushroom fricasee
Flavorful skirt steak marinated in Asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. $16.95
* We are required to warn you that undercooked food is a health risk.

FIGURE 12-8. The menu after a few minor font-size changes to the headings

Font Weight (Boldness)

After font families and size, the remaining font properties are straightforward. For example, if you want a text element to appear in bold, use the **font-weight** property to adjust the boldness of the type.

font-weight

Values: normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900

Default: normal

Applies to: All elements

Inherits: Yes

As you can see, the **font-weight** property has many predefined values, including descriptive terms (**normal**, **bold**, **bolder**, and **lighter**) and nine numeric values (**100** to **900**) for targeting various weights of a font if they are available. Because most fonts commonly used on the web have only two weights, normal (or Roman) and bold, the font weight value you will use the most is **bold**. You may also use **normal** to make text that would otherwise appear in bold (such as strong text or headlines) appear at a normal weight.

The numeric values come in handy when using variable fonts with a large range of weights. In fact, some Google web fonts require numeric size values. FIGURE 12-9 shows the range of weights available for the variable Open Sans font. By contrast, if multiple weights are not available, as for the Times example in the figure, numeric settings of **600** and higher generally result in bold text (although even that can vary by browser).

If a separate bold face is not available, the browser may "synthesize" a bold font by beefing up the available normal face (see **Note**).

NOTE

CSS Fonts Module Level 3 introduced the **font-synthesis** *property, which allows authors to turn off (with a value of* **none***) or allow synthesized bold fonts (value of* **weight***).*

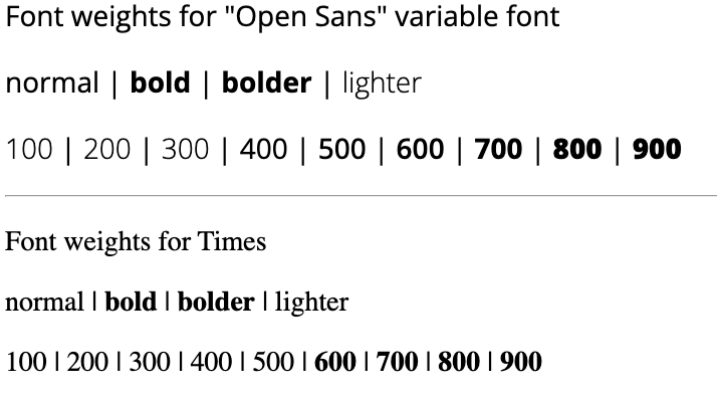

Font weights for "Open Sans" variable font
normal | bold | bolder | lighter
100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900
Font weights for Times
normal | bold | bolder | lighter
100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900

FIGURE 12-9. The effect (and lack thereof!) of **font-weight** values. Notice that the numeric values are much more useful for the variable font.

Font Style (Italics)

The **font-style** property affects the **posture** of the text—that is, whether the letter shapes are vertical (**normal**) or slanted (**italic** and **oblique**).

font-style

Values: normal | italic | oblique *angle*

Default: normal

Applies to: All elements

Inherits: Yes

The **font-style** property is most often used to make text **italic**, but another common use is to make text that is italicized in the browser's default styles (such as emphasized text) display as **normal**. The **oblique** value specifies a slanted version of the font; browsers generally display **oblique** exactly the same as **italic**, but CSS Fonts Module Level 4 added the ability to customize the display further by specifying a specific slant angle between –90deg and 90deg.

You can practice adjusting font weights and styles in EXERCISE 12-3.

EXERCISE 12-3. Making text bold and italic

Back to the menu. I've decided that I'd like all of the menu item names to be in bold text. What I'm not going to do is wrap each one in **<b>** tags…that would be so 1996! I'm also not going to mark them up as **strong** elements, as that's not semantically accurate. Instead, the right thing to do is simply apply a style to the semantically correct **dt** (definition term) elements to make them all bold at once. Add this rule to the end of the embedded style sheet, save the file, and see how it looks in your browser:

```
dt { font-weight: bold; }
```

Now that all the menu item names are bold, some of the text I've marked as **strong** isn't standing out very well, so I'll make them italic for further emphasis. To do this, simply apply the **font-style** property to the **strong** element:

```
strong { font-style: italic;}
```

Once again, save and reload. It should look like FIGURE 12-10.

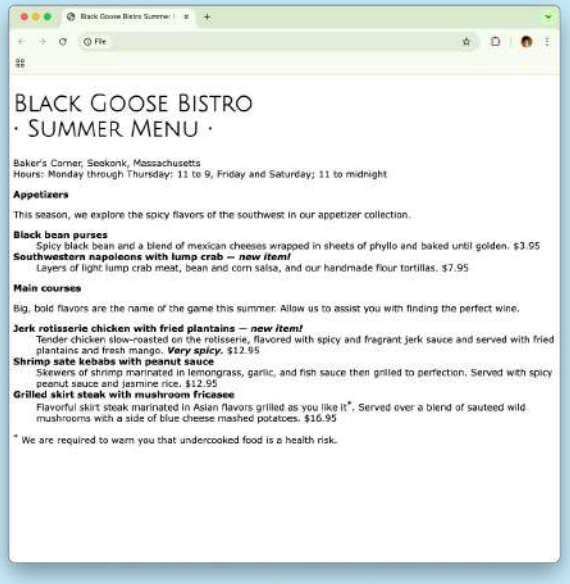

BLACK GOOSE BISTRO
· SUMMER MENU ·
Baker's Corner, Seakonk, Massachusetts
Hours: Monday through Thursday: 11 to 9, Friday and Saturday: 11 to midnight
Appetizers
This season, we explore the spicy flavors of the southwest in our appetizer collection.
Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden. $3.95
Southwestern napoleons with lump crab — new item!
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. $7.95
Main courses
Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.
Jerk rotisserie chicken with fried plantains — new item!
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. Very spicy. $12.95
Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice. $12.95
Grilled skirt steak with mushroom fricasee
Flavorful skirt steak marinated in Asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. $16.95
* We are required to warn you that undercooked food is a health risk.

FIGURE 12-10. Applying the **font-weight** and **font-style** properties

Small Caps

`font-variant`

Values: `normal` | `small-caps`

Default: `normal`

Applies to: All elements

Inherits: Yes

Some typefaces come in a "small caps" variant. This is a separate font design that uses small uppercase-style letters in place of lowercase letters. Small-caps characters are designed to match the size and density of lowercase text so they blend in.

Small caps should be used for strings of three or more capital letters appearing in the flow of text, such as acronyms and abbreviations, that may look jarring as full-sized capitals. Compare NASA and USA in the standard font to NASA and USA in small caps. Small caps are also recommended for dates and times, like 1AM or 2024AD, and of course they can be used purely for stylistic effect.

When the **`font-variant`** property was introduced in CSS2.1, it was a one-trick pony that allowed designers to specify a small-caps font for text elements. CSS Fonts Module Level 3 has greatly expanded the role of **`font-variant`** (see the upcoming sidebar **"Advanced Typography Features"**), but for now we're looking at only the CSS2.1 version.

In most cases, browsers simulate small caps by scaling down uppercase letters in the current font. To typography sticklers, this is less than ideal as it results in inconsistent stroke weights, but you may find it an acceptable option for adding variety to small amounts of text. You will see an example of small caps when we use the **`font-variant`** property in EXERCISE 12-5.

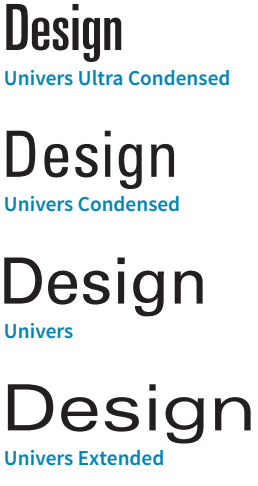


FIGURE 12-11. Examples of condensed, normal, and extended (expanded) versions of the Univers typeface

Font Stretch (Condensed and Expanded)

`font-stretch`

Values: `normal` | `ultra-condensed` | `extra-condensed` | `condensed` | `semi-condensed` | `semi-expanded` | `expanded` | `extra-expanded` | `ultra-expanded`

Default: `normal`

Applies to: All elements

Inherits: Yes

The **`font-stretch`** property tells the browser to select a normal, condensed, or expanded font in the font family (FIGURE 12-11). If the browser cannot find a matching font, it will *not* try to synthesize the width by stretching or squeezing text; it may just substitute a font of a different width.

The Shorthand font Property

Specifying multiple font properties for each text element can get repetitive and lengthy, so the creators of CSS provided the shorthand **font** property, which compiles all the font-related properties into one rule.

font

Values: *font-style font-weight font-variant font-stretch font-size/line-height font-family* | caption | icon | menu | message-box | small-caption | status-bar

Default: Depends on default value for each property listed

Applies to: All elements

Inherits: Yes

The value of the **font** property is a list of values for all the font properties we just looked at, separated by character spaces. It is important to note that only the CSS2.1 version of **font-variant** (**small-caps**) can be used in the **font** shortcut. In this property, the order of the values is important:

```
{ font: style weight stretch variant size/line-height family; }
```

At minimum, the **font** property *must* include a **font-size** value and a **font-family** value, in that order. Omitting one or putting them in the wrong order causes the entire rule to be invalid. This is an example of a minimal **font** property value:

```
p { font: 1.25rem sans-serif; }
```

Once you've met the size and family requirements, the other values are optional and may appear in any order *prior* to the **font-size**. When style, weight, stretch, or variant values are omitted, its value is set to **normal**. As mentioned in the **Warning**, that makes it easy to accidentally override a previous setting with the shorthand property, so be careful when you use **font**.

There is one value in there, **line-height**, that you have not seen yet. As its name suggests, it adjusts the height of the text line and is used to add space between lines of text (this property is covered in more detail in **"Text Line Adjustments,"** later in this chapter). It appears just after **font-size**, separated by a slash, as shown in these examples:

```
h3 { font: oblique bold small-caps 1.5em/1.8em Verdana, sans-serif; }

h2 { font: bold 1.75em/2 sans-serif; }
```

In EXERCISE 12-4, we'll use the shorthand **font** property to make some changes to the **h1** headings in the Bistro's menu.

System font keywords

The **font** property also has a number of keyword values (**caption**, **icon**, **menu**, **message-box**, **small-caption**, and **status-bar**) that represent system fonts, the fonts used by operating systems for things like labels for icons and menu

WARNING

Be careful when using shorthand properties like **font**. *Any omitted property resets to its default value, which can surprise you by undoing a value you set earlier. On the flip side, using a shorthand property is a good way to get a blank slate if you need one.*

TIP

The **font-stretch** property only gained browser support after 2019. If you need to support very old browsers (like Internet Explorer) and are using the **font-stretch** property in your **font** shorthand, provide a fallback declaration that omits the **font-stretch** value. You will end up with two **font** shorthands, like this:

```
h3 {
  font: bold 1.25em Helvetica;
  font: bold expanded 1.25em
Helvetica;
}
```

EXERCISE 12-4.
Using the shorthand font property

One last tweak to the menu, and then we'll take a brief break. To save space, we can specify multiple font properties for the **h1** element with one declaration with the shorthand **font** property:

```
h1 {
  font: 400 2.5rem "Julius Sans One", Verdana, sans-serif;
}
```

Now the **font-family** and **font-size** are in one rule. I've also included the numeric weight setting that Google recommends for the Julius Sans One font (400) here, instead of using a separate **font-weight** rule.

Save the page and take a look at it in the browser. If you've done your job right, it should look nearly the same as in the previous step, except the lines in the heading will be slightly more slender.

TIP

Here's a little tip when using the **font** shorthand with an **h1** or other element that the browser makes bold by default. If you want to be sure the heading stays bold, you need to explicitly include the **bold** keyword in the **font** values. This is because if you omit a shorthand value, it resets to the default for the property according to the CSS specification (**normal**), not the browser's default value (**bold**). Shorthand properties can be tricky that way…pay attention so you don't leave something out and accidentally override a default or inherited value you were counting on. This is not an issue for our menu design because we're using a light font for the heading.

items. These may be useful when you're designing a web application so that it matches the user's working environment. They're considered shorthand values because they encapsulate the font, size, style, and weight used for each purpose in a single keyword.

Let's get some practice using the shorthand **font** property. Like **font** itself, EXERCISE 12-4 is short and sweet.

We've finally made our way through the various properties for controlling fonts in CSS (it took a while!)—but that's just one aspect of text presentation. Before we get to the remainder of the text-related properties, let's take a little side trip and learn about a few more types of selectors that will give us more flexibility in targeting elements in the document for styling.

DESCENDANT, ID, AND CLASS SELECTORS

So far, we've been using element names as selectors. In the last chapter, you also saw how to group selectors together in a comma-separated list so you can apply properties to several elements at once. Here are examples of the two types of selectors you already know:

Element selector	`p { color: navy; }`
Grouped selectors	`p, ul, td, th { color: navy; }`

The disadvantage of selecting elements this way is that the property—in this case, navy-blue text—is applied to every paragraph (and each instance of the other listed elements) in the document. Sometimes you want to apply a rule to only a particular paragraph or paragraphs. In this section, we'll look at three selector types that allow us to do just that: descendant selectors, ID selectors, and class selectors.

Descendant Selectors

A **descendant selector** targets elements that are contained within (and therefore are descendants of) another element. It is an example of a **contextual selector** because it selects the element based on its context or relation to another element. The sidebar **"Other Contextual Selectors"** lists some others.

Descendant selectors are indicated in a list with the items separated by character spaces. This example targets emphasized text (**em**) elements, but *only* when they appear in list items (**li**); emphasized text in paragraphs and other elements will be unaffected (FIGURE 12-12):

```
li em { color: olive; }
```

Advanced Typography Features

If sophisticated typography is important to you and supports the business goals of the sites you design, you may want to get familiar with the advanced typography properties introduced in CSS Fonts Module Levels 3 and 4. The following is a quick overview of the properties and what they do, to give you an idea of what's possible.

Font-variant- properties

The **font-variant-** prefixed properties aim to give designers and developers access to special characters (**glyphs**) in fonts that can make the typography on a page more sophisticated. The old **font-variant** property that has been around since the beginning of CSS has been upgraded to be a shorthand property for all of the properties listed here. You can still use it today with the original **small-caps** value, and it will be perfectly valid.

font-variant-ligatures

Provides a way to control the use of ligatures on web pages. A **ligature** is a glyph that combines two or more characters into one symbol. One common example is the combination of a lowercase *f* and *i*, where the dot on the *i* becomes part of the *f* (fi). Ligatures can smooth out the appearance of known awkward letter pairings, and ligature glyphs are included in many fonts.

font-variant-caps

Allows the selection of small-cap glyphs (**small-caps**) from the font's character set rather than simulating them in the browser.

font-variant-position

Selects superscript (**super**) or subscript (**sub**) glyphs from the font's character set when they are available. This property is poorly supported and is at risk of being dropped in Level 4.

font-variant-numeric

Allows selection between various number character styles if they are available.

font-variant-alternates

Provides a way to specify alternate glyphs, for fonts that offer more than one glyph for a particular character—for example, a few swash designs for the letter S, or an old-fashioned s that looks more like an f.

font-variant-east-asian

Allows selection of particular Asian glyphs.

Additional properties

In addition to the **font-variant-** collection, these newer properties give you more control over typography:

font-size-adjust

The size we perceive text to be on the page often has more to do with the height of the lowercase *x* (its **x-height**) than the specified size of the text. For example, 10-point type with a relatively large x-height is likely easier to read than 10-point type with dainty little lowercase letters (compare the appearance of the 1rem fonts in FIGURE 12–8). The **font-size-adjust** property allows the browser to adjust the size of a fallback font until its x-height matches the x-height of the first-choice font. This can ensure better legibility even when a fallback font needs to be used.

font-kerning

Kerning is the space between character glyphs. Fonts typically contain metadata about which letter pairs need to be cozied up together to make the spacing in a word look consistent. The **font-kerning** property allows the font's kerning information to be applied (**normal**), turned off (**none**), or left to the browser's discretion (**auto**).

font-feature-settings

This property affects special characters such as swashes, small caps, ligatures, automatic fractions, and more; however, it is recommended that authors use the appropriate **font-variant-** properties instead.

font-language-override

This experimental property controls the use of language-specific glyphs.

FURTHER READING

For a nice overview of OpenType features and why they are worthwhile, read "Caring About OpenType Features" by Tim Brown at Typekit Practice (*oreil.ly/M48jW*).

In a descendant selector, a character space between element names means that the second element must be contained within the first.

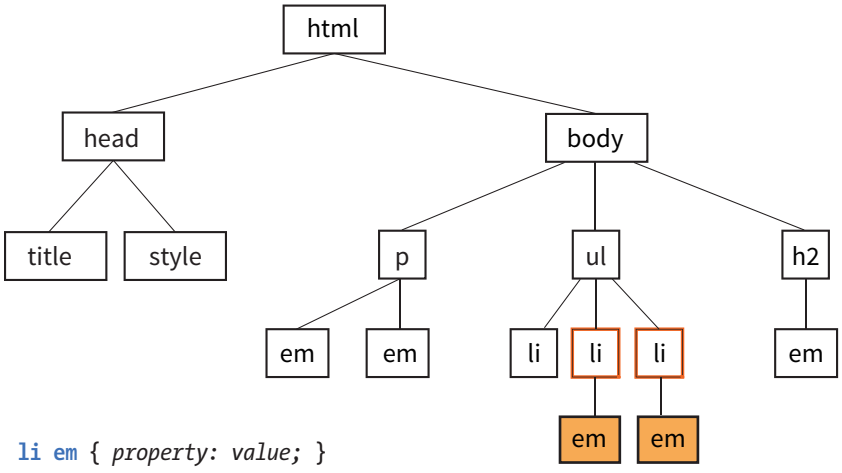


FIGURE 12-12. Only `em` elements within `li` elements are selected. The other `em` elements are unaffected.

Other Contextual Selectors

Descendant selectors are one of five types of contextual selectors (called **combinators** in spec). The most commonly used are child selectors, next-sibling selectors, and subsequent-sibling selectors. (The **column combinator** (||) used for selecting cells in a table or grid is not supported and at risk of being dropped.)

Don't let their appearance in a sidebar fool you; these are very useful selectors to have in your toolkit.

Child selector

A **child selector** is similar to a descendant selector, but it targets only the direct children of a given element. There may be no other hierarchical levels in between. Child selectors are indicated with the greater-than symbol (>). The following rule affects emphasized elements, but only when they are directly contained in a **p** element; an **em** element inside a link (**a**) within the paragraph will not be affected:

```
p > em {font-weight: bold;}
```

Next-sibling selector

A **next-sibling selector** targets an element that comes directly after another element with the same parent. It is indicated with a plus (+) sign. This rule gives special treatment to paragraphs that immediately follow an **h1**; other paragraphs are unaffected:

```
h1 + p {font-style: italic;}
```

Subsequent-sibling selectors

A **subsequent-sibling selector**, indicated with a tilde (~), selects an element that shares a parent with the specified element and occurs after it in the source order. They do not need to follow one another directly. The following rule selects any **h2** that both shares a parent element (such as a **section** or **article**) with an **h1** and appears after it in the document:

```
h1 ~ h2 {font-weight: normal;}
```

Here's another example that shows how contextual selectors may be grouped in a comma-separated list. This rule targets **em** elements, but only when they appear in **h1**, **h2**, or **h3** headings:

```
h1 em, h2 em, h3 em { color: red; }
```

It is also possible to nest descendant selectors several layers deep. This example targets **em** elements that appear in anchors (**a**) in ordered lists (**ol**):

```
ol a em { font-variant: small-caps; }
```

ID Selectors

Back in **Chapter 5**, you learned about the **id** attribute, which gives an element a unique identifying name (its **id** reference). The **id** attribute can be used with any element, and it is commonly used to give meaning to the generic **div** and **span** elements. ID selectors allow you to target elements by their **id** values. The symbol that identifies ID selectors is the octothorpe (**#**), also known as a hash or pound symbol.

The # symbol identifies an ID selector.

Here is an example of a list item with an **id** reference:

```
<li id="sleestak">Sleestak T-shirt</li>
```

You can write a style rule just for that list item by using an ID selector, like so (notice the **#** preceding the **id** reference):

```
li#sleestak { color: olive; }
```

Because **id** values must be unique in the document, it is safe to omit the element name. The following rule is equivalent to the last one:

```
#sleestak { color: olive; }
```

You can also use an ID selector as part of a contextual selector. In this example, a style is applied only to **a** elements that appear within the element with an **id** of "resources". In this way, you can treat links in the element named "resources" differently from all the other links on the page without any additional markup.

```
#resources a { text-decoration: none; }
```

You should be beginning to see the power of selectors and how they can be used strategically along with well-planned semantic markup.

Class Selectors

The other element identifier you learned about in **Chapter 5** is the **class** identifier, used to classify elements into a conceptual group. Unlike the **id** attribute, multiple elements may share a **class** name. Not only that, but an element may belong to more than one class. Classes are overwhelmingly the most popular way of styling most elements.

The Universal Selector

The universal element selector (*) matches any element, like a wildcard in programming languages. The style rule

```
* {border: 1px solid gray;}
```

puts a 1-pixel gray border around every element in the document. It is also useful as a contextual selector, as shown in this example that selects all elements in an "intro" section:

```
#intro * { color: gray; }
```

Be aware that every element will be selected with the universal selector, including some that you might not be expecting to style. For example, some styles might mess up your form controls, so if your page contains form inputs, the safest bet is to avoid the universal selector.

You can target elements belonging to the same class with—you guessed it—a **class selector**. Class names are indicated with a period (`.`) at the beginning of the selector. For example, to select all paragraphs with `class="special"`, use this selector (the period indicates the following word is a class selector):

```
p.special { color: orange; }
```

To apply a property to *all* elements of the same class, omit the element name in the selector (be sure to leave the period; it's the character that indicates a class). This example targets all paragraphs and any other element that has been marked up with `class="special"`:

```
.special { color: orange; }
```

SPECIFICITY 101

In **Chapter 3, Some Big Concepts You Need to Know**, I introduced you to the term **specificity**, the idea that more specific selectors have more weight when it comes to settling style rule conflicts. Now that you know a few more selectors, it's a good time to revisit this very important concept.

This list of selector types from most to least specific should serve you well in most scenarios:

- **Inline styles** with the `style` attribute are more specific than (and will override...)
- **ID selectors**, which are more specific than (and will override...)
- **Class selectors**, which are more specific than (and will override...)
- **Individual element selectors**

The full story is more complicated, but if you aren't sure which style rule will prevail in a conflict, you can use the three-box technique for calculating the specificity of a selector (there are other techniques out there too, but I like this one). Start by drawing three boxes. For each selector, count up the number of IDs in the selector, and put that number in the first box. Next, count up the number of classes and pseudo-classes (see **Note**) in the selector, and put that number in the second box. Finally, count up the element names, and put that number in the third box:

```
[ ID ]  [ classes ]  [ elements ]
```

Specificity is compared box by box, from left to right. The first box that is not a tie determines which selector (and rule) wins. Here is a simple example of two conflicting rules for the `h1` element:

```
h1 { color: red;}              [0] [0] [1]
h1.special { color: lime; }    [0] [1] [1]
```

The second one has a class selector and the first one doesn't; therefore, the second one is more specific and has more weight.

NOTE

Only styles marked as `!important` *will override inline styles. Although* `!important` *has nothing to do with specificity, it does affect which style sheet source has priority, and therefore is a player in deciding which style rule wins. It is considered bad practice to use* `!important` *only to override specificity.*

NOTE

A pseudo-class selector targets an element when it is in a certain state—for example ,when the cursor is hovering over it (`:hover`*). They are indicated by the colon (*`:`*) character. You will learn about pseudo-classes in* **Chapter 14, Selectors**.

How about something more complicated? Which of these two selectors has more specificity?

```
.x.x.x.x.x.x.x.x  a:link                                  [0] [9] [1]
article#main aside.sidebar:hover > h1:first-of-type       [1] [3] [3]
```

The first selector has a string of eight class names (represented by **.x**) and one pseudo-class (**:link**), so it gets a 9 in the classes box. But the second selector contains an ID (**#main**), so it has a 1 in the ID box (whereas the first rule had a 0) and is therefore more specific.

You may occasionally do a full specificity calculation, but in most cases you'll have a feel for which selector is more specific by following previously listed general guidelines. Even better, you should avoid overly complicated selectors in general, keeping the structure flat and the specificity low to avoid having to deal with messy selectors and calculations.

You can use specificity strategically to keep your style sheets simple and your markup minimal. For example, it is possible to set a style for an element (**p**, in this example), and then override it by using more specific selectors:

```
p { line-height: 1.2em; }              [0] [0] [1]
blockquote p { line-height: 1em; }     [0] [0] [2]
p.intro { line-height: 2em; }          [0] [1] [1]
```

In these examples, **p** elements that appear within a **blockquote** have a smaller line height than ordinary paragraphs. However, all paragraphs with a **class** of "intro" will have a 2em line height, even if they appear within a **blockquote**, because class selectors are more specific.

Understanding the concepts of inheritance and specificity is critical to mastering CSS, and there is a lot more to be said about specificity. The **"More About Specificity"** sidebar provides useful references.

Now, back to the menu. Fortunately, our Black Goose Bistro page has been marked up thoroughly and semantically, so we have a lot of options for selecting specific elements. Give these new selectors a try in EXERCISE 12-5.

More About Specificity

The specificity overview in this chapter is enough to get you started, but when you get more experienced and your style sheets become more complicated, you may find that you need a more thorough understanding of the inner workings.

- For the technical explanation of exactly how specificity is calculated, see the Selectors Level 4 specification at *w3.org/TR/selectors4/#specificity*.
- The article "The Specifics on CSS Specificity" by Chris Coyier has a more detailed overview of the method for calculating specificity: *css-tricks.com/specifics-on-css-specificity/*.
- Eric Meyer provides a thorough yet more digestible description of this system in his book *Selectors, Specificity, and the Cascade: Applying CSS to Documents* (O'Reilly). This material is also included in his book coauthored with Estelle Weyl, *CSS: The Definitive Guide, 5e* (O'Reilly).
- As for most web design topics, the MDN Web Docs site provides a comprehensive explanation: *oreil.ly/emg3P*.

EXERCISE 12-5. Using selectors

Let's add a few more style rules using descendant, ID, and class selectors. In addition to font-related properties, we'll also throw in some color using the **color** property.

You may be wondering where I got the color names in the exercises. I'm not making them up! They are defined in CSS Color Module Level 3, and I've posted the full collection with their RGB equivalents at *learningwebdesign.com/colornames.html*. There are more sophisticated ways to specify color that I'll cover in *Chapter 13, Colors and Backgrounds*. For now, we'll have fun with the standard color names.

1. Let's add some attention-getting color to the "new item!" text next to certain menu item names. They are marked up as **strong**, so we can apply the **color** property to the **strong** element. Add this property, save the file, and reload it in the browser:

   ```
   strong {
     font-style: italic;
     color: tomato;
   }
   ```

   That worked, but now the **strong** content "Very spicy" in the description is "tomato" red too, and that's not what we want. The solution is to use a contextual selector that targets only the **strong** elements that appear in **dt** elements. Remove the **color** declaration you just wrote from the **strong** rule, and create a new rule that targets only the **strong** elements within definition list terms:

   ```
   dt strong { color: tomato; }
   ```

2. Look at the document source, and you will see that the content has been divided into three unique **div**s: **info**, **appetizers**, and **entrees**. We can use these to our advantage when it comes to styling. For now, let's do something simple and apply the color "cadetblue" to the text in the **div** with the **id** "info". Because color inherits, we only need to apply the property to the **div**, and it will be passed down to the **h1** and **p** elements:

   ```
   #info { color: cadetblue; }
   ```

3. Now let's get a little fancier and make the paragraph inside the "info" section italic in a way that doesn't affect the other paragraphs on the page. Again, a contextual selector is the answer. This rule selects only paragraphs contained within the "info" section of the document:

   ```
   #info p { font-style: italic; }
   ```

4. Now let's give special treatment to all of the prices on the menu. Fortunately, they have all been marked up with **span** elements:

   ```
   <span class="price">$3.95</span>
   ```

   So all we have to do is write a rule using a class selector to make the prices italic and gray:

   ```
   .price {
     font-style: italic;
     color: gray;
   }
   ```

5. Similarly, in the "info" **div**, we can change the appearance of the **span**s that have been marked up as belonging to the "label" class to make the labels stand out:

   ```
   .label {
     font-variant: small-caps;
     font-style: normal;
   }
   ```

6. Finally, let's style the warning at the bottom of the page. We'll make it small and "tomato" red. It has been given the **class** "warning", so we can use that as a selector to target just that paragraph. While we're at it, let's apply the same style to the **sup** element (the footnote asterisk) earlier on the page so they match. We can use a grouped selector so we don't need to write separate rules:

   ```
   p.warning, sup {
     font-size: small;
     color: tomato;
   }
   ```

FIGURE 12-13 shows the results of all these changes. We now have a few touches of color and special typography treatments, and you've gotten to try out some font properties and selector types—but we're not done yet. There are more text properties to tackle.

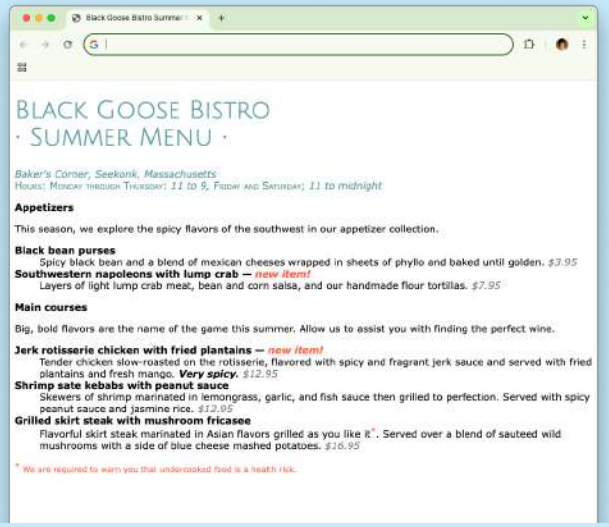

BLACK GOOSE BISTRO
· SUMMER MENU ·
Baker's Corner, Seekonk, Massachusetts
Hours: Monday through Thursday 11 to 9, Friday and Saturday 11 to midnight
Appetizers
This season, we explore the spicy flavors of the southwest in our appetizer collection.
Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden. $3.95
Southwestern napoleons with lump crab — new item!
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. $7.95
Main courses
Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.
Jerk rotisserie chicken with fried plantains — new item!
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. Very spicy. $12.95
Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice. $12.95
Grilled skirt steak with mushroom fricasee
Flavorful skirt steak marinated in Asian flavors grilled as you like it*. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes. $16.95
* We are required to warn you that undercooked food is a health risk.

FIGURE 12-13. The current state of the bistro menu

TEXT LINE ADJUSTMENTS

The next batch of text properties have to do with the treatment of whole lines of text rather than the shapes of characters. They allow authors to format web text with indents, extra space between lines (leading), and different horizontal alignments.

Line Height

line-height

Values: *number* | *length measurement* | *percentage* | `normal`

Default: `normal`

Applies to: All elements

Inherits: Yes

The **`line-height`** property defines the minimum distance from baseline to baseline in text. We saw it earlier as part of the shorthand **`font`** property. The **`line-height`** property is said to specify a "minimum" distance because if you put a tall image or large characters on a line, the height of that line expands to accommodate them.

A **baseline** is the imaginary line upon which the bottoms of characters sit. Setting a line height in CSS is similar to adding leading in traditional typesetting; however, instead of space being added between lines, the extra space is split above and below the text. The result is that **`line-height`** defines the height of a **line-box** in which the text line is vertically centered (FIGURE 12-14).

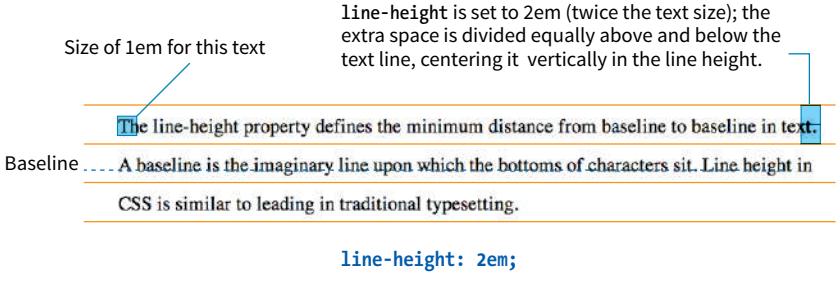

Size of 1em for this text
line-height is set to 2em (twice the text size); the extra space is divided equally above and below the text line, centering it vertically in the line height.
The line-height property defines the minimum distance from baseline to baseline in text.
Baseline
A baseline is the imaginary line upon which the bottoms of characters sit. Line height in
CSS is similar to leading in traditional typesetting.
line-height: 2em;

FIGURE 12-14. Text lines are centered vertically in the line height.

These examples show three different ways to make the line height twice the height of the font size:

```
p { line-height: 2; }
p { line-height: 2em; }
p { line-height: 200%; }
```

When a number is specified alone, as shown in the first example, it acts as a scaling factor that is multiplied by the current font size to calculate the **line-height** value.

Line heights can also be specified in one of the CSS length units. Ems and percentage values are based on the current font size of the element. In these three examples, if the font size is 16 pixels, the calculated line height would be 32 pixels.

The difference between using a scaling factor (number value) and a relative value (em or a percentage) is how they inherit. If you set the line height with a scaling factor for a whole document on the **body** element, its descendants inherit the *multiplier*. For example, if the scaling factor is set to 2 for the **body**, a 24-pixel headline will end up with a line height of 48 pixels.

If you set the **line-height** on an element using ems or percentages, its descendants inherit the *calculated* size based on the body's font size. For example, if the line height is set to 1em for the **body** element (calculated at 16 pixels), a 24-pixel headline inherits the calculated 16-pixel line height, not the 1em value. This is likely not the effect you are after, making numeric scaling values a more intuitive option.

Indents

The **text-indent** property indents the first line of text by a specified amount.

text-indent

Values: *length measurement* | *percentage*

Default: 0

Applies to: Block containers

Inherits: Yes

You can specify a length measurement or a percentage value for **text-indent**. Here are a few examples (the results are shown in FIGURE 12-15):

```
p#1 { text-indent: 2em; }
p#2 { text-indent: 25%; }
p#3 { text-indent: -35px; }
```

Percentage values are calculated based on the width of the *parent* element, and they are passed down to their descendant elements as percentage values (not calculated values). So, if a **div** has a **text-indent** of 10%, so will all of its descendants.

In the third example, notice that a negative value was specified. That's just fine; it will cause the first line of text to hang out to the left of the left text edge (also called a **hanging indent**).

NOTE

The **text-indent** *property indents just the first line of a block. If you want space along the whole side of the text block, use one of the* **margin** *or* **padding** *properties to add it.*

Designers may be accustomed to specifying indents and margins in tandem, but to be consistent with how CSS handles them, margins will be discussed as part of the box model in **Chapter 14, Selectors**.

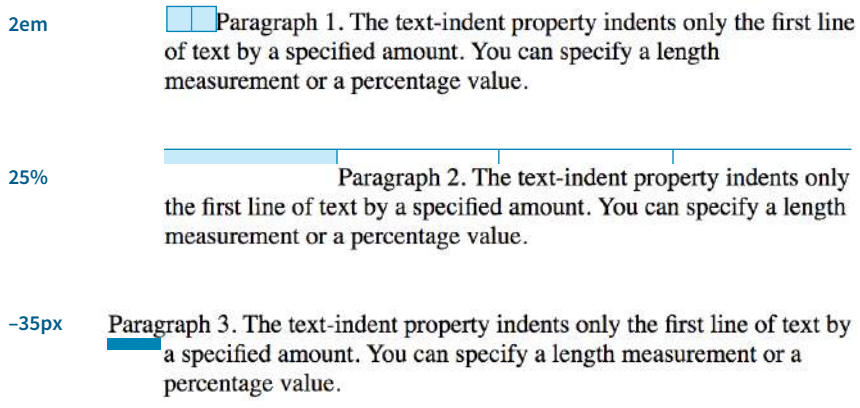

2em
Paragraph 1. The text-indent property indents only the first line of text by a specified amount. You can specify a length measurement or a percentage value.
25%
Paragraph 2. The text-indent property indents only the first line of text by a specified amount. You can specify a length measurement or a percentage value.
–35px
Paragraph 3. The text-indent property indents only the first line of text by a specified amount. You can specify a length measurement or a percentage value.

FIGURE 12-15. Examples of the `text-indent` property

> **TIP**
>
> If you use a hanging indent, be sure that there is also a left padding applied to the element. Otherwise, the hanging text may disappear off the left edge of the browser window.

Horizontal Text Alignment

You can align text for web pages just as you would in a word processing or desktop publishing program with the **`text-align`** property.

text-align

Values: `left` | `right` | `center` | `justify` | `start` | `end`

Default: `start`

Applies to: Block containers

Inherits: Yes

This is a fairly straightforward property to use. The results of the various CSS2.1 **`text-align`** values are shown in FIGURE 12-16:

`text-align: left`	Aligns text on the left margin
`text-align: right`	Aligns text on the right margin
`text-align: center`	Centers the text in the text block
`text-align: justify`	Aligns text on both right and left margins

CSS Text Module Level 3 added the **`start`** and **`end`** values, which specify the side of the line box the text should align to (see **Note**). This accommodates languages that are written vertically and right to left. For left-to-right reading languages, **`start`** corresponds to **`left`**.

Good news—only five more text properties to go! Then we'll be ready to try a few of them in the Black Goose Bistro menu.

NOTE

CSS Text Module Level 3 also defines two new properties related to text alignment: **`text-align-last`** *(for aligning the last line of text) and* **`text-justify`** *(for more fine-tuned control over how space is inserted in justified text).*

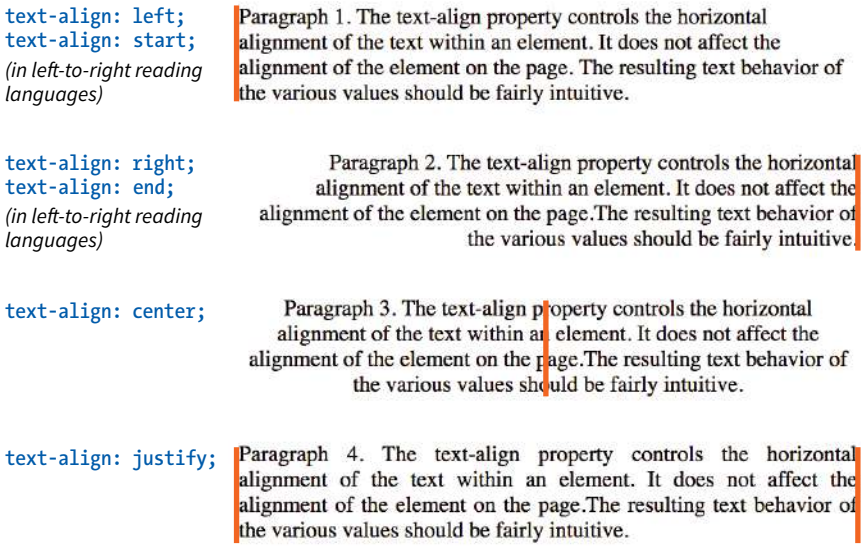

text-align: left;
text-align: start;
(in left-to-right reading languages)
Paragraph 1. The text-align property controls the horizontal alignment of the text within an element. It does not affect the alignment of the element on the page. The resulting text behavior of the various values should be fairly intuitive.
text-align: right;
text-align: end;
(in left-to-right reading languages)
Paragraph 2. The text-align property controls the horizontal alignment of the text within an element. It does not affect the alignment of the element on the page.The resulting text behavior of the various values should be fairly intuitive.
text-align: center;
Paragraph 3. The text-align property controls the horizontal alignment of the text within an element. It does not affect the alignment of the element on the page.The resulting text behavior of the various values should be fairly intuitive.
text-align: justify;
Paragraph 4. The text-align property controls the horizontal alignment of the text within an element. It does not affect the alignment of the element on the page.The resulting text behavior of the various values should be fairly intuitive.

FIGURE 12-16. Examples of CSS2.1 `text-align` values

UNDERLINES AND OTHER "DECORATIONS"

If you want to put a line under, over, or through text, or if you'd like to turn off the underline under links, then **`text-decoration`** is the property for you (see **Note**).

text-decoration

Values: none | underline | overline | line-through | blink

Default: none

Applies to: All elements

Inherits: No, but since lines are drawn across child elements, they may look like they are "decorated" too

The values for **`text-decoration`** are intuitive and are shown in FIGURE 12-17:

`underline`	Underlines the element
`overline`	Draws a line over the text
`line-through`	Draws a line through the text

One of the most popular uses of the **`text-decoration`** property is turning off the underlines that appear automatically under linked text, as shown here:

```
a { text-decoration: none; }
```

NOTE

The `text-decoration` *property as it is shown here represents the universally supported syntax from the CSS2.1 specification. In CSS Text Decoration Module Level 3 (w3.org/TR/css-text-decor-3/), the* `text-decoration` *property becomes a shorthand representing* `text-decoration-line`, `text-decoration-color`, *and* `text-decoration-style`. *As of this writing, only* `text-decoration-line` *(with the same values as the CSS2.1* `text-decoration`*) has significant browser support, so for the time being, stick with* `text-decoration` *as discussed here or provide it as a fallback. The Level 3 module also adds the related* `text-underline-position` *property with the values* `auto`, `under`, `left`, *and* `right`. *See the specification for more information.*

There are a few cautionary words to be said regarding **text-decoration**:

- First, if you get rid of the underlines under links, be sure there are other cues to compensate, such as color and weight.
- On the flip side, because underlines are such a strong visual cue to "click here," underlining text that is *not* a link may be misleading and frustrating. Consider whether italics may be an acceptable alternative.
- If you looked closely at the property listing, you'll see the value **blink**, but there is no reason to make your text blink on and off. Browser makers agree and therefore have dropped support for blinking text, and the value is deprecated in the spec.

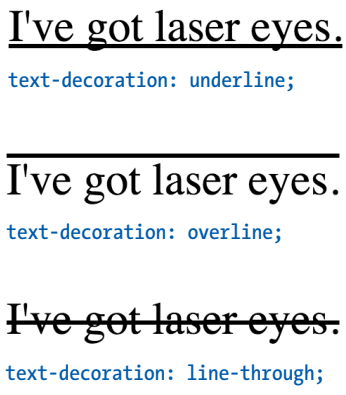


FIGURE 12-17. Examples of **text-decoration** values

CHANGING CAPITALIZATION

Many word processing and desktop publishing programs have a feature that allows you to change the capitalization of text with the press of a button. This feature makes it easy to see how headlines might look in all capital letters without needing to retype them. CSS includes this feature as well with the **text-transform** property.

text-transform

Values: none | capitalize | lowercase | uppercase | full-width

Default: none

Applies to: All elements

Inherits: Yes

When you apply the **text-transform** property to a text element, it changes its capitalization when it renders without changing the way it is typed in the source. The values are as follows; FIGURE 12-18 shows the results of the most commonly used options:

none	As it is typed in the source
capitalize	Capitalizes the first letter of each word
lowercase	Makes all letters lowercase
uppercase	Makes all letters uppercase
full-width	Chooses a "full-width" version of a character if one exists (not well supported)
full-size-kana	Converts small Japanese Kana characters to full-size

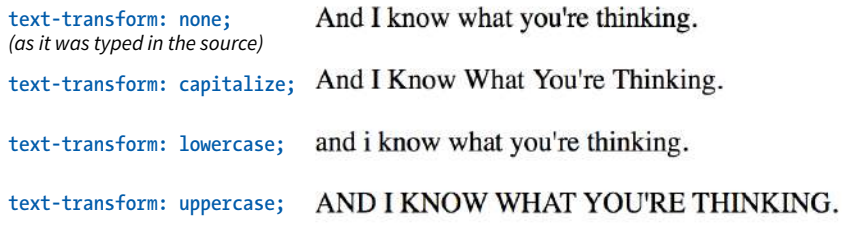

text-transform: none;
(as it was typed in the source)
And I know what you're thinking.
text-transform: capitalize;
And I Know What You're Thinking.
text-transform: lowercase;
and i know what you're thinking.
text-transform: uppercase;
AND I KNOW WHAT YOU'RE THINKING.

FIGURE 12-18. The `text-transform` property changes the capitalization of characters when they are displayed, regardless of how they are typed in the source.

SPACED OUT

The next two text properties are used to insert space between letters (**`letter-spacing`**) or words (**`word-spacing`**) when the text is displayed.

letter-spacing

Values: *length measurement* | `normal`

Default: `normal`

Applies to: All elements

Inherits: Yes

word-spacing

Values: *length measurement* | `normal`

Default: `normal`

Applies to: All elements

Inherits: Yes

The **`letter-spacing`** and **`word-spacing`** properties do just what they say: add space between the letters of the text or words in a line, respectively.

FIGURE 12-19 shows the results of letter spacing and word spacing applied to the simple paragraph shown here:

```
<p>Black Goose Bistro Summer Menu</p>
```

FIGURE 12-19. Adjusting `letter-spacing` (top) and `word-spacing` (bottom)

It is worth noting that when you specify em measurements, the calculated size is passed down to child elements, even if they have a smaller font size than the parent.

In EXERCISE 12-6 later in this chapter, we'll make one last trip back to the Black Goose Bistro menu and use the **letter-spacing** property on **h2**s.

TEXT SHADOW

The **text-shadow** property adds a "shadow" below your text that makes it seem to hover or pop out above the page. Now that flat-color design has become the fashion, drop shadows aren't as popular as they once were; however, they can still be a useful visual tool, particularly when your text is in front of a patterned or photographic background. Text shadows are drawn behind the text but in front of the background and border, if there is one.

text-shadow

Values: *horizontal-offset vertical-offset blur-radius color* | none

Default: none

Applies to: All elements

Inherits: Yes

The value for the **text-shadow** property is two or three measurements (a horizontal offset, a vertical offset, and an optional blur radius) and a color. FIGURE 12-20 shows the result of the following minimal text shadow declaration:

```
h1 {
  color: darkgreen;
  text-shadow: .2em .2em silver;
}

h1 {
  color: darkgreen;
  text-shadow: -.3em -.3em silver;
}
```

FIGURE 12-20. A minimal text drop shadow

The first value is a horizontal offset that positions the shadow to the right of the text (a negative value pulls the shadow to the *left* of the text). The second measurement is a vertical offset that moves the shadow down by the specified amount (a negative value moves the shadow *up*). The declaration ends with the color specification. If the color is omitted, the text color will be used.

That should give you an idea of how the first two measurements work, but that sharp shadow doesn't look very...well...shadowy. What it needs is a blur radius measurement to soften it up. Zero (0) is no blur, and the blur gets softer with higher values (FIGURE 12-21). Usually, you just have to fiddle with values until you get the effect you want.

FIGURE 12-21. Adding a blur radius to a text drop shadow

It is possible to apply several text shadows to the same element. If you vary the position and blur amounts, you can give the text the appearance of multiple light sources.

So, go have some fun with text shadows, but be careful not to overdo it. Not only can drop shadows make text difficult to read, but adding a shadow to everything can slow down page performance (scrolling, mouse interactions, etc.) as well, which is particularly problematic for mobile browsers. In addition, be careful that your text doesn't require a shadow in order to be visible. My advice is to use drop shadows as an enhancement in a way that isn't critical if they don't appear.

EXERCISE 12-6 gives you a chance to try out more text formatting properties to put a little polish on the Black Goose Bistro menu.

The Other Text Properties

In the interest of saving space and keeping this an introductory-level book, I haven't given these properties the full treatment, but they are worth knowing about. Each is labeled with the CSS level in which it was introduced.

For even more text-related properties in development, see the following CSS modules:

- CSS Text Module Level 3: *w3.org/TR/css-text-3*
- CSS Text Decoration Module Level 3: *w3.org/TR/css-text-decor-3*
- CSS Text Module Level 4 (still in Working Draft and considered experimental): *w3.org/TR/css-text-4*

vertical-align (CSS2) Specifies the vertical alignment of an inline element's baseline relative to the baseline of the surrounding text. It is also used to set the vertical alignment of content in a table cell (**td**).

white-space (CSS2) Specifies how whitespace in the element source is handled in layout. For example, the **pre** value preserves the character spaces and returns found in the source, similar to the **pre** HTML element.

hanging-punctuation (Level 3) Determines whether the punctuation mark may be outside the element's line box at the start or end of a line. Hanging punctuation can make margins appear tidier.

hyphens (Level 3) Provides control over how text is hyphenated. **manual** means hyphenation happens only when there is a hyphen added in the source, **auto** gives control to the browser, and **none** turns off hyphenation completely.

overflow-wrap (Level 3) Specifies whether browsers are allowed to break words to fit text in its bounding box.

tab-size (Level 3) Specifies the length of the tab character (Unicode code point 0009) in number of characters or a length measurement.

text-align-last (Level 3) Specifies how the last line of a block of text should be justified when the **text-align** property on the element is set to **justify**. For example, it is often preferable to have the last line left-justified for justified text to avoid awkwardly spaced words.

text-justify (Level 3) Specifies the manner in which space is to be added within and between words when the **text-align** property on the element is set to **justify**.

word-break and **line-break** (Level 3) Affects how text wrapping is calculated within words and lines, respectively, in various languages, including East Asian languages (Chinese, Japanese, Korean).

line-padding (Level 4) Adjusts space around inline elements at the start or end of a line.

text-autospacing (Level 4) Controls space between particular classes of characters; for example, numerals and punctuation.

text-spacing-trim (Level 4) Controls spacing around Chinese, Japanese, and Korean punctuation characters.

word-boundary-detection (Level 4) Gives authors control of where word boundaries are. This is used for soft-wrapping lines of text and may be useful for internationalization.

EXERCISE 12-6. Finishing touches

Let's add a few finishing touches to the Bistro's menu. It might be useful to save the file and look at it in the browser after each step to see the effect of your edits. The finished style sheet is provided in the *materials* folder for this chapter.

1. First, I have a few global changes to the **body** element in mind. I've had a change of heart about the **font-family**. A serif font such as Georgia would be more sophisticated and appropriate for a bistro menu. Let's also use the **line-height** property to open up the text lines and make them easier to read. Make these updates to the **body** style rule:

```
body {
  font-family: Georgia, serif;
  line-height: 1.75;
}
```

2. I'd also like the **h1** heading to be a different color from the rest of the "info" section. Add this new **color** declaration to the existing **h1** rule. Even though the **h1** is inside the "info" section and inherits the "cadetblue" color we assigned to it earlier, its element selector has more weight than the ID selector used for the **div**, so the new color value ("olivedrab") wins.

```
h1 {
  font: bold 1.5rem "Marko One", Georgia, serif;
  color: olivedrab;
}
```

3. Now, to imitate a fancy restaurant menu, we're going to center a few key elements on the page with the **text-align** property. Write a rule with a grouped selector to center the headings and the "info" section:

```
h1, h2, #info {
  text-align: center;
}
```

4. I want to make the "Appetizer" and "Main Courses" **h2** headings more eye-catching too. Instead of large, bold type, I'm going to use all uppercase letters, extra letter spacing, and color to call attention to the headings. Here's the new rule for **h2** elements that includes all of these changes:

```
h2 {
  font-size: 1rem;
  text-transform: uppercase;
  letter-spacing: .5em;
  color: cadetblue;
}
```

5. We're really close now; just a few more tweaks to those paragraphs that appear right after the **h2** headings. Let's center those too and make them italic:

```
h2 + p {
  text-align: center;
  font-style: italic;
}
```

   Note that I've used a next-sibling selector (**h2 + p**) to select any paragraph that follows an **h2**.

6. Next, add a softer color to the menu item names (in the existing **dt** rule). I've chosen "olivedrab" so they match the main page headline. Note that the **strong** elements in those **dt** elements stay "tomato" red because the color applied to the **strong** elements overrides the color inherited by their parents.

```
dt {
  font-weight: bold;
  color: olivedrab;
}
```

7. Finally, for kicks, add a light drop shadow under the **h1** heading using the pale gray color "gainsboro". Notice how small the shadow values are—a little goes a long way! You can play around with the values a little to see how it works. Honestly, I don't think it's doing any favors for the design, but I did want you to have an opportunity try out the property.

```
h1 {
  font: bold 1.5em "Marko One", Georgia, serif;
  color: olive;
  text-shadow: .1em .1em .05em gainsboro;
}
```

And we're done! FIGURE 12-22 shows how the menu looks now—a big improvement over the unstyled version, and we used only text and color properties to do it. Notice that we didn't touch a single character of the document markup in the process. That's the beauty of keeping style separate from structure.

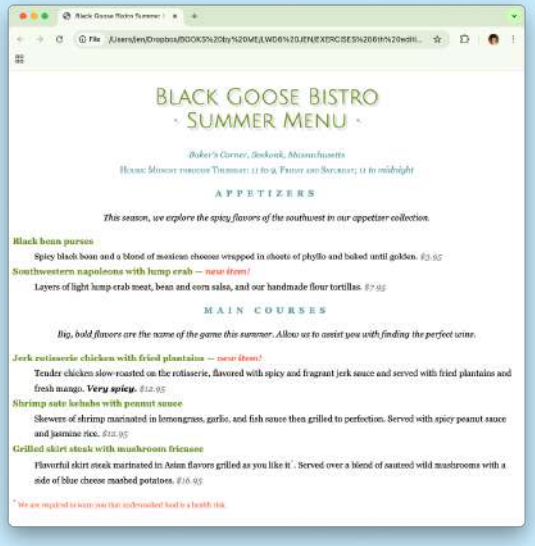

BLACK GOOSE BISTRO
SUMMER MENU
APPETIZERS
This season, we explore the spicy flavors of the southwest in our appetizer collection.
Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden.
Southwestern napoleons with lump crab — new item!
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas.
MAIN COURSES
Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.
Jerk rotisserie chicken with fried plantains — new item!
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. Very spicy.
Shrimp satè kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice.
Grilled skirt steak with mushroom ricasse
Flavorful skirt steak marinated in Asian flavors grilled as you like it. Served over a blend of sauteed wild mushrooms with a side of blue cheese mashed potatoes.

FIGURE 12-22. The formatted Black Goose Bistro menu

CHANGING LIST BULLETS AND NUMBERS

NOTE

This section documents the CSS2.1 `list-style` *types that are supported by all current browsers. The CSS Counter Styles Level 3 recommendation (***w3.org/TR/css-counter-styles-3***) extends the marker functionality significantly, including a method for authors to define their own list styles and allowing for numbering in many languages. It's beyond the scope of this book, but it's worth knowing there are more options for marker customization.*

Before I close out this chapter on text properties, I want to show you a few tweaks you can make to bulleted and numbered lists. As you know, browsers automatically insert bullets before unordered list items and numbers before items in ordered lists (the **list markers**). For the most part, the rendering of these markers is determined by the browser. However, CSS provides a few properties that allow authors to choose the type and position of the marker, or turn them off entirely.

Choosing a Marker

Browsers automatically generate markers for ordered and unordered lists, but you can use apply the `list-style-type` property to the `ul`, `ol`, or `li` element to select the type of marker that appears before each list item (see **Note**).

Boost Your Typography Skills

The menu we've been working on in the exercises isn't bad, considering it uses only a few CSS properties, but it's certainly not an example of killer graphic design. Unfortunately, typography and graphic design lessons are outside of the scope of this book, where our focus is on writing code with the help of some simplified designs. However, it is absolutely possible to create amazing typography using CSS and web fonts.

If you search for "examples of beautiful web typography" you'll find many galleries bursting with possibilities and inspiration, such as *awwwards.com/websites/typography/* (FIGURE 12-23). If you'd like to feel more confident about typography, you may want to sign up for a typography course. You will find several free tutorials online, including these:

- **Design & Code Better Web Typography** (*betterwebtype.com*), a free email newsletter–based course developed by Matej Latin. Matej has also written a book, *Better Web Typography for a Better Web*, which you can find a link to on the same page.
- **Typography for Developers** (*freecodecamp.org/news/how-to-design-good-typography/*), a free two-hour video tutorial from freeCodeCamp's Beau Carnes.

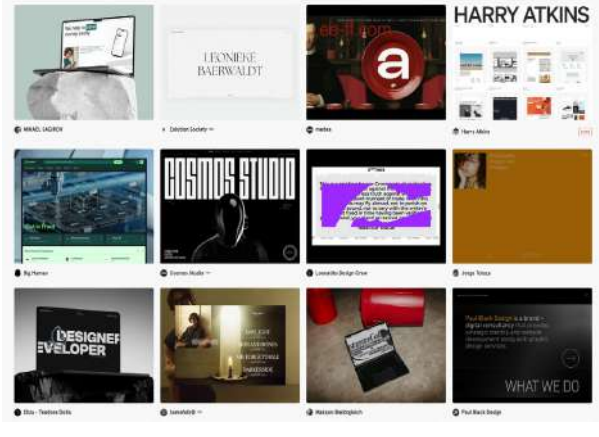


FIGURE 12-23. *awwwards.com/websites/typography/* a frequently updated gallery of inspiring typography examples

list-style-type

Values: none | disc | circle | square | decimal | decimal-leading-zero | lower-alpha | upper-alpha | lower-latin | upper-latin | lower-roman | upper-roman | lower-greek

Default: disc

Applies to: ul, ol, and li (or elements whose display value is list-item)

Inherits: Yes

More often than not, developers use the **list-style-type** property with its value set to **none** to remove bullets or numbers altogether. This is handy when you're using list markup as the foundation for a horizontal navigation menu or the entries in a web form. You can keep the semantics but get rid of the pesky markers.

The **disc**, **circle**, and **square** values generate bullet shapes just as browsers have been doing since the beginning of the web itself (FIGURE 12-24). Unfortunately, there is no way to change the appearance (size, color, etc.) of generated bullets, so you're stuck with the browser's default rendering.

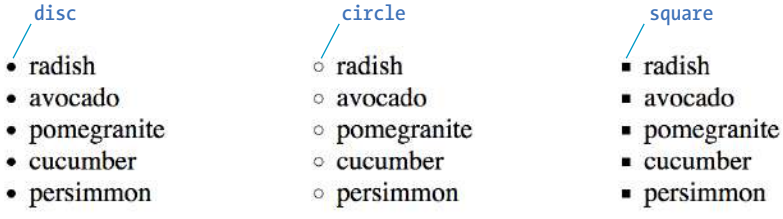


FIGURE 12-24. The **list-style-type** values **disc**, **circle**, and **square**

The remaining keywords (TABLE 12-1) specify various numbering and lettering styles for use with ordered lists.

TABLE 12-1. Lettering and numbering system (CSS2.1)

Keyword	System
decimal	1, 2, 3, 4, 5...
decimal-leading-zero	01, 02, 03, 04, 05...
lower-alpha	a, b, c, d, e...
upper-alpha	A, B, C, D, E...
lower-latin	a, b, c, d, e... (same as **lower-alpha**)
upper-latin	A, B, C, D, E... (same as **upper-alpha**)
lower-roman	i, ii, iii, iv, v...
upper-roman	I, II, III, IV, V...
lower-greek	α, β, γ, δ, ε...

List Item Display Role

You may have noticed that the list style properties apply to "elements whose display value is **list-item**." The CSS2.1 specification allows any element to perform like a list item if its **display** property is set to **list-item**. For example, you could automatically bullet or number a series of paragraphs by setting the **display** property of paragraph (**p**) elements to **list-item**, as shown in this example:

```
article p {
  display: list-item;
  list-style-type: upper-alpha;
}
```

Marker Position

By default, the marker hangs outside the content area for the list item, displaying as a hanging indent. The **list-style-position** property allows you to pull the bullet inside the content area so it runs into the list content.

list-style-position

Values: inside | outside | hanging

Default: outside

Applies to: ul, ol, and li (or elements whose display value is list-item)

Inherits: Yes

I've applied a light-blue background color to the list items in FIGURE 12-25 to reveal the boundaries of their content area boxes.

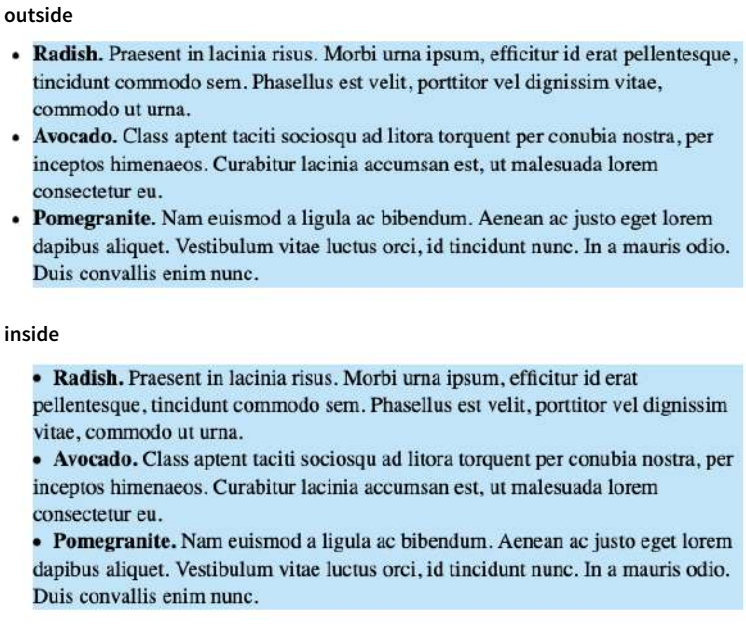

outside
Radish. Praesent in lacinia risus. Morbi urna ipsum, efficitur id erat pellentesque, tincidunt commodo sem. Phasellus est velit, porttitor vel dignissim vitae, commodo ut urna.
Avocado. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Curabitur lacinia accumsan est, ut malesuada lorem consectetur eu.
Pomegranite. Nam euismod a ligula ac bibendum. Aenean ac justo eget lorem dapibus aliquet. Vestibulum vitae luctus orci, id tincidunt nunc. In a mauris odio. Duis convallis enim nunc.
inside
Radish. Praesent in lacinia risus. Morbi urna ipsum, efficitur id erat pellentesque, tincidunt commodo sem. Phasellus est velit, porttitor vel dignissim vitae, commodo ut urna.
Avocado. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Curabitur lacinia accumsan est, ut malesuada lorem consectetur eu.
Pomegranite. Nam euismod a ligula ac bibendum. Aenean ac justo eget lorem dapibus aliquet. Vestibulum vitae luctus orci, id tincidunt nunc. In a mauris odio. Duis convallis enim nunc.

FIGURE 12-25. The **list-style-position** property

You can see that when the position is set to **outside** (top), the markers fall outside the content area. When it is set to **inside** (bottom), the markers are tucked into the content area.

```
li {background-color: #c2e4f8;}
ul#outside {list-style-position: outside;}
ul#inside {list-style-position: inside;}
```

The **hanging** value for **list-style-position** is similar to **inside**, but the markers appear outside and abutting the left edge of the shaded area.

Make Your Own Bullets

You can use your own image as a bullet by using the **list-style-image** property (see also the **"Styling Markers"** sidebar).

list-style-image

Values: url(*location*) | none

Default: none

Applies to: ul, ol, and li (or elements whose display value is list-item)

Inherits: Yes

The value of the **list-style-image** property is the URL of the image you want to use as a marker. The **list-style-type** is set to **disc** as a backup in case the image does not display or the property isn't supported by the browser or other user agent. Here's an example; the result is shown in FIGURE 12-26:

```
ul {
  list-style-type: disc;
  list-style-image: url(/images/rainbow.gif);
  list-style-position: outside;
}
```

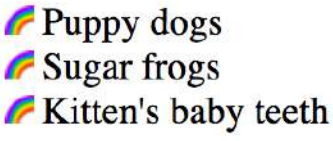


FIGURE 12-26. Using an image as a marker

Shorthand list-style property

There is a **list-style** shorthand property that combines the values for type position, and image, in any order. For example:

```
ul {
  list-style: url(/images/rainbow.gif) disc outside;
}
```

As with all shorthands, be careful not to override list style properties set earlier in the style sheet.

Styling Markers

It is possible to specify a text character to be used as a marker and apply certain styles to it (**font-size** and **color** are the best supported as of this writing) with the **::marker** selector. This is an example of a pseudo-element selector, which will be discussed in **Chapter 14**. The MDN Web Docs site has a clear introduction (*oreil.ly/icCTv*).

FINISHING UP TEXT STYLES

Wow! What a chapter! We started by looking at properties for specifying fonts and character shapes, followed by a review of all the text-level settings and effects. You also got to use descendant, ID, and class selectors and looked a little more closely at specificity. We topped it off with the properties available for adding some style to lists. I don't expect you to have all of these properties committed to memory (although many will become second nature the more you practice), but let's see how you do on the following questions.

TEST YOURSELF

It's time to see how well you understand the font properties and selectors introduced in this chapter. Check **Appendix A** for the answers if you get stuck.

1. Match the style properties with the text samples in FIGURE 12-27.
   a. _____ `{font-size: 1.5em;}`
   b. _____ `{text-transform: capitalize;}`
   c. _____ `{text-align: right;}`
   d. _____ `{font-family: Verdana, sans-serif; font-size: 1.5em;}`
   e. _____ `{letter-spacing: 3px;}`
   f. _____ `{font: bold italic 1.2em Verdana, sans-serif;}`
   g. _____ `{text-transform: uppercase;}`
   h. _____ `{text-indent: 2em;}`
   i. _____ `{font-variant: small-caps;}`

Look for the good in others and they'll see the good in you.
default font and size

1 Look For The Good In Others And They'll See The Good In You.

2 Look for the good in others and they'll see the good in you.

3 Look for the good in others and they'll see the good in you.

4 Look for the good in others and they'll see the good in you.

5 Look for the good in others and they'll see the good in you.

6 LOOK FOR THE GOOD IN OTHERS AND THEY'LL SEE THE GOOD IN YOU.

7 Look for the good in others and they'll see the good in you.

8 LOOK FOR THE GOOD IN OTHERS AND THEY'LL SEE THE GOOD IN YOU.

9 ***Look for the good in others and they'll see the good in you.***

FIGURE 12-27. Styled text samples

2. Here's a chance to get a little practice writing selectors. Using the diagram shown in FIGURE 12-28, write style rules that make each of the elements described here red (`color: red;`). Write the selectors as efficiently as possible.

   a. All text elements in the document

   b. `h2` elements

   c. `h1` elements and all paragraphs

   d. Elements belonging to the class `special`

   e. All elements in the "intro" section

   f. `strong` elements in the "main" section

   g. Extra credit: just the paragraph that appears after an `h2`

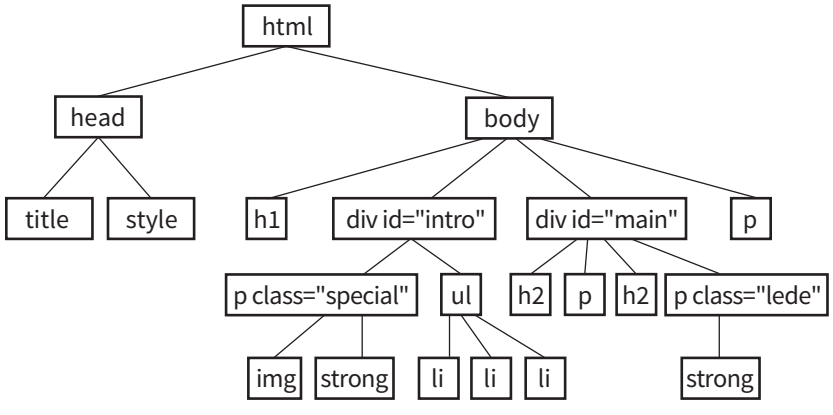


FIGURE 12-28. Sample document structure

3. Using the three-box method, calculate the specificity of each of the following selectors:

   a. `header h1`

   b. `h1.hero`

   c. `dt + dd`

   d. `main #byline h2.author`

   e. `main > p.lede`

4. Which selector in question 3 has the most weight? Why?

CSS REVIEW: FONT AND TEXT PROPERTIES

In this chapter, we covered the properties used to format text elements. Here is a summary, in alphabetical order.

Property	Description
`direction`	Indicates whether the text reads left to right or right to left
`font`	A shorthand property that combines font properties
`font-family`	Specifies a typeface or generic font family
`font-feature-settings`	Allows access to lesser-used OpenType features
`font-kerning`	Controls how browsers implement kerning data (space between characters)
`font-language-override`	Controls use of language-specific glyphs
`font-size`	Specifies the size of the font
`font-size-adjust`	Matches the x-height of a fallback font with the specified font
`font-stretch`	Selects a condensed, normal, or expanded font
`font-style`	Specifies italic or oblique fonts
`font-synthesis`	Controls whether a browser may simulate bold or italic fonts
`font-variant`	Specifies a small-caps font
`font-variant-alternates`	Selects alternate versions of character glyphs
`font-variant-caps`	Selects small caps and similar alternates when available
`font-variant-east-asian`	Selects alternate glyphs in Chinese, Japanese, and Korean
`font-variant-ligatures`	Selects ligatures for certain letter pairs when available
`font-variant-numeric`	Selects alternate number glyphs
`font-variant-position`	Selects subscript or superscript character glyphs
`font-weight`	Specifies the boldness of the font
`hanging-punctuation`	Indicates whether the punctuation may hang outside the content box
`hyphens`	Controls how text is hyphenated
`letter-spacing`	Inserts space between letters
`line-break`	Describes rules for breaking lines
`line-height`	Indicates the distance between baselines of neighboring text lines

Property	Description
`list-style-image`	Specifies an image to be used as a list marker
`list-style-position`	Puts a list marker inside or outside the content area
`list-style-type`	Selects the marker type for list items
`overflow-wrap`	Specifies whether the browser can break lines within words to prevent overflow
`tab-size`	Specifies the length of a tab character
`text-align`	Indicates the horizontal alignment of text
`text-align-last`	Specifies how the last line in justified text is aligned
`text-decoration`	Specifies underlines, overlines, and lines through
`text-indent`	Specifies the amount of indentation of the first line in a block
`text-justify`	Denotes how space is distributed in justified text
`text-shadow`	Adds a drop shadow under the text
`text-transform`	Changes the capitalization of text when it displays
`unicode-bidi`	Works with Unicode bidirectional algorithms
`vertical-align`	Adjusts the vertical position of inline elements relative to the baseline
`white-space`	Specifies how whitespace in the source is displayed
`word-break`	Specifies whether to break lines within words
`word-spacing`	Inserts space between words
`word-wrap`	Indicates whether the browser can break lines within words to prevent overflow (same as **`overflow-wrap`**)

CHAPTER
13

COLORS AND BACKGROUNDS

If you had seen the web back in 1993, you would have found it to be a dreary affair by today's standards—every background was gray, and all the text was black. Then came Netscape Navigator and, with it, a handful of HTML attributes that allowed rudimentary (but welcome) control over font colors and backgrounds. For years, we made do. But thankfully, we now have style sheet properties that have laid those unmentionable presentational attributes to rest.

We're going to cover a *lot* of ground in this chapter. You'll get familiar with some color terminology that will provide important context for understanding how to specify color values in CSS. Of course, I'll introduce you to all of the properties for adding colors to the foreground and background of elements, as well as how to apply repeating background patterns. Finally, we'll take a look at linear and radial gradient patterns. Let's get started by getting to know how color works in general.

IN THIS CHAPTER

Color spaces and models

CSS color names

RGB, HSL, and OKLCH color values

Alpha transparency

Foreground and background colors

Tiling background images

Color gradients

COLOR SPACES AND MODELS

The average human eye can see an enormous spectrum of colors. Color is subjective, but there are instances when you need to communicate a specific color, whether on a screen, in print, or in other contexts. A **color space** provides a way to define and reproduce colors, whether analog (like Pantone colors or paint swatches) or in the digital realm using numeric values.

A system for specifying colors numerically is called a **color model**. These are the options you see when selecting colors from a color picker in a graphics program or browser. Some examples should help here. The grayscale color space defines colors according to one numeric parameter: the percentage of brightness. Colors in the RGB color space are defined by three number values:

amounts of red, green, and blue light. In the HSL color model, the three number values represent hue, saturation, and lightness. CMYK colors used in printing are defined by four parameters: percentages of cyan, magenta, yellow, and black ink. You'll sometimes see color models represented as 3D models such as spheres, cubes, or cylinders (like the HSL figure later in this chapter), depending on the number and relationship of axes in the model.

The range of colors a color model can represent is called its **gamut**. Because the web is a digital medium presented on screens, colors are displayed in an RGB color space. Specifically, the web uses a standardized color space called sRGB (Standard RGB), which is a subset of the full RGB spectrum (see the **"RGB vs. sRGB vs. AdobeRGB"** sidebar for more details). sRGB has become the de facto color space for computer monitors, scanners, printers, and digital devices typically used by businesses and consumers. There are other color spaces defined within the full RGB spectrum that have wider gamuts than sRGB. These traditionally have been used for digital photography and color printing, where creators don't want to compromise on the color range in their images; however, as monitors improve, wide-gamut models like OKLCH are becoming relevant for web design as well.

FIGURE 13-1 shows a simplified comparison of the gamuts of a few RGB-based color spaces as compared to the color range of human sight.

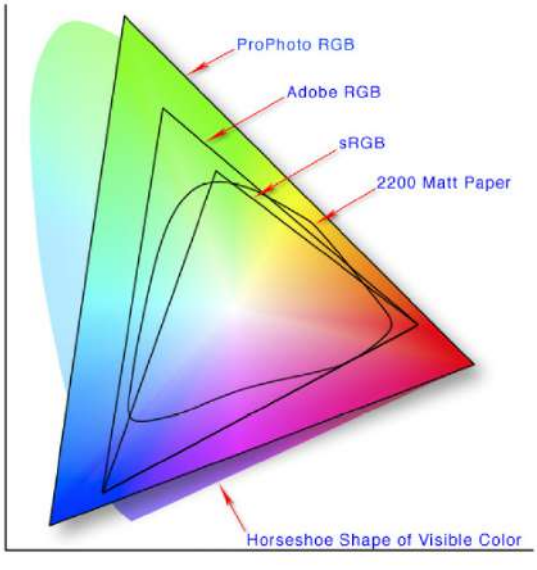


FIGURE 13-1. A comparison of common color spaces compared to the range of colors detectable to the average human eye [Image credit: Color space. (2023, December 7). In Wikipedia: *en.wikipedia.org/wiki/Color_space*. CC BY 2.5 license.]

The CSS Working Group has been working overtime on more sophisticated and future-facing ways to specify color. Much of their work relates to trying to keep ahead of advances in screen technology. The screens on our devices

FUN FACT

Color monitors in the early days of the web had a gamut of only 256 colors. Designers often used a "web-safe palette" made up of 216 system colors shared by Windows and macOS in order to control color consistency across platforms. Ah, those were the days. You may still come across options for "web palette" or "web-safe colors" in image editing applications, but there is no point in limiting the color palette for today's displays.

have gamuts as well. Newer monitors, such as Display P3 monitors, are capable of displaying wider gamuts with brighter colors than the sRGB displays we've been using for the last decade or so.

CSS Color Module Level 4 (*w3.org/TR/css-color-4/*) introduces ways to specify colors in those wider-gamut RGB spaces, as wider-gamut monitors are becoming increasingly popular and affordable. The newly available color values in the Level 4 module can also accommodate specialized use cases, including nonweb color applications. I'll give you the rundown on all the new color spaces in the sidebar **"Advanced Color Features,"** later in this chapter. For now, the takeaway is that color spaces are not created equal and that the latest CSS Color Module includes methods for specifying them all.

RGB vs. sRGB vs. AdobeRGB

I've mentioned that the RGB color space creates colors with combinations of red, green, and blue light (the **primary** light colors). "RGB" is somewhat of a generic term for the colors available on a device or in its software, however. In practice, there are multiple color spaces that can be used to describe RGB colors.

How color spaces work

What is it that differentiates all of the RGB-based color spaces that we saw in FIGURE 13–1 from one another? The short answer is, it depends on what their definitions of "red, "green," and "blue" are! Let's dig into that answer a bit more deeply.

Let's start with a little history. In the 1930s, color experts mapped the entire spectrum of colors that can be seen by the typical human eye and standardized it as the CIE 1931 XYZ color space. "XYZ" refers to the three axes on which color wavelengths are mapped, creating a 3D space representing all points of visible light. This is often used as a reference for other color spaces.

Now for a bit of screen technology. Every pixel on a monitor or device screen is made up of three subpixels, each displaying exclusively some amount of red, blue, or green light. When the light from the pixel hits your eyes, you perceive the combination of those three primaries as a specific color. But there are many different shades of red, green, and blue. The industry needed to standardize what "red" should be for red subpixels, what "green" should be, and so on, so that everyone was speaking the same RGB language. They did this by selecting coordinates on the XYZ axes of the CIE color space for each primary. For example, on devices and software that use the sRGB color space, red is set at x: .64 , y: .33, z: .2126. The purpose of standardizing the red, green, and blue points, of course, is consistency of color reproduction across devices. When you connect the points between the primaries, you get the gamut of that RGB-based color space, as shown in FIGURE 13–1.

RGB standards

sRGB and AdobeRGB are two examples of standardized gamuts for displays and software.

The sRGB color space was standardized in a joint effort by Hewlett-Packard (HP) and Microsoft in 1996 and is based on monitors and HDTVs of that time. It is the official color space of the web, but it has also been adopted by consumer-oriented devices that capture, display, or print digital images.

As useful as the sRGB color space is for most digital tasks, its gamut is fairly limited. Adobe created its own standardized RGB color space that sets the red, blue, and green points to encompass a much larger gamut, preserving some brilliant reds and greens that are impossible to achieve in sRGB. Adobe's initial intent was to bridge the large gap between colors in the sRGB color space and the wider CMYK color space used for print. Many cameras and video devices today allow you to capture images in AdobeRGB. If you display those images on an sRGB monitor, every color gets converted to its nearest sRGB equivalent, and you won't see the full spectrum of color available in the image. However, if you have a wide-gamut monitor, you will see the color as it was originally recorded.

Specifying RGB values

It's worth mentioning that regardless of the RGB-based color space you're in, you specify RGB colors according to the RGB color model with three color values—one for red, one for green, one for blue—as I'll cover later in the chapter. A pure red is always represented as R:100%, G:0%, B:0%, but how bright that red appears depends on the color space you're working in.

As a web designer, you can stick with the default sRGB color space. But it is good to know that there are color spaces with wider gamuts out there for specialized purposes and that the W3C has broadened the CSS Color Module to include those spaces and any additional color models or spaces to come.

SPECIFYING COLOR VALUES

Now that I've gotten the technical talk out of the way, let's get to the practical bits. There are two main ways to specify colors in style sheets. The first way is with a predefined color name, as we have been doing so far in this book:

```
color: red;
color: olive;
color: blue;
```

The second, and far more common, way is with numeric values according to a particular color model. When you open a color picker in an image program, you usually see a lot of color model options, but for the web, the most popular color models are RGB and HSL. Here are a few examples to give you a preview of numeric color value syntax:

```
color: #FF0000;
color: rgb(256 0 0);
color: hsl(265 50% 10%);
```

We'll get to all the ins and outs of numeric color values in a moment, but first, a short and sweet section on the standard color names.

Color Names

The most intuitive way to specify a color is to call it by name. Unfortunately, you can't make up just any color name and expect it to work. It has to be one of the 140 (rather fanciful) color keywords predefined in the CSS Color Module. You can specify names like "burlywood," "peachpuff," "oldlace," and my long-time favorite, "papayawhip"! The colors are shown in FIGURE 13-2, but if you want a more accurate view, point your browser at *learningwebdesign.com/colornames.html*. There is also a `transparent` keyword that can be used with any property that takes a color value to make it fully transparent.

> **FUN FACT**
>
> The set of 140 color names, also known as the X11 color names, were originally provided with the X Window System for Unix. In prior versions of CSS, there were only 17 colors to choose from.

Color names are simple to use—just drop one into place as the value for any color-related property:

```
color: rosybrown;
background-color: rebeccapurple;
border-bottom-color: lightcoral;
```

Names are easy, but as you can see, they are limited. By far, the most common way to specify a color is using numeric values for colors in the RGB and HSL color models. We'll look at both here, as well as the up-and-coming OKLCH color model that can specify brighter colors on wide-gamut displays.

aliceblue 240,248,255 #F0F8FF	antiquewhite 250,235,215 #FAEBD7	aqua 0,255,255 #00FFFF	aquamarine 127,255,212 #7FFFD4	azure 240,255,255 #F0FFFF	beige 245,245,220 #F5F5DC	bisque 255,228,196 #FFE4C4	black 0,0,0 #000000
blanchedalmond 255,255,205 #FFFFCD	blue 0,0,255 #0000FF	blueviolet 138,43.226 #8A2BE2	brown 165,42,42 #A52A2A	burlywood 222,184,135 #DEB887	cadetblue 95,158,160 #5F9EA0	chartreuse 127,255,0 #7FFF00	chocolate 210,105,30 #D2691E
coral 255,127,80 #FF7F50	cornflowerblue 100,149,237 #6495ED	cornsilk 255,248,220 #FFF8DC	crimson 220,20,60 #DC143C	cyan 0,255,255 #00FFFF	darkblue 0,0,139 #00008B	darkcyan 0,139,139 #008B8B	darkgoldenrod 184,134,11 #B8860B
darkgray 169,169,169 #A9A9A9	darkgreen 0,100,0 #006400	darkkhaki 189,183,107 #BDB76B	darkmagenta 139,0,139 #8B008B	darkolivegreen 85,107,47 #556B2F	darkorange 255,140,0 #FF8C00	darkorchid 153,50,204 #9932CC	darkred 139,0,0 #8B0000
darksalmon 233,150,122 #E9967A	darkseagreen 143,188,143 #8FBC8F	darkslateblue 72,61,139 #483D8B	darkslategray 47,79,79 #2F4F4F	darkturquoise 0,206,209 #00CED1	darkviolet 148.0,211 #9400D3	deeppink 255,20,147 #FF1493	deepskyblue 0,191,255 #00BFFF
dimgray 105,105,105 #696969	dodgerblue 30,144255 #1E90FF	firebrick 178,34,34 #B22222	floralwhite 255,250,240 #FFFAF0	forestgreen 34,139,34 #229B22	fuchsia 255,0,255 #FF00FF	gainsboro 220,220,220 #DCDCDC	ghostwhite 248,248,255 #F8F8FF
gold 255,210,0 #FFD700	goldenrod 218,165,32 #DAA520	gray 128,128,128 #808080	green 0,128,0 #008000	greenyellow 173,255,47 #ADFF2F	honeydew 240,255,240 #F0FFF0	hotpink 255,105,180 #FF69B4	indianred 205,92,92 #CD5C5C
indigo 75,0,130 #4B0082	khaki 240,230,140 #F0D58C	lavender 230,230,250 #E6E6FA	lavenderblush 255,240,245 #FFF0F5	lawngreen 124,252,0 #7CFC00	lemonchiffon 255,250,205 #FFFACD	lightblue 173,216,230 #ADD8E6	lightcoral 240,128,128 #F08080
lightcyan 224,255,255 #F0FFFF	lightgoldenrodyellow 250,250,210 #FAFAD2	lightgreen 144,138,144 #90EE90	lightgray 211,211,211 #D3D3D3	lightpink 255,182,193 #FFB6C1	lightsalmon 255,160,122 #FFA07A	lightseagreen 32,178,170 #20B2AA	lightskyblue 135,206,250 #87CEFA
lightslategray 119,136,153 #778899	lightsteelblue 176,196,222 #B0C4DE	lightyellow 255,255,224 #FFFFE0	lime 0,255,0 #00FF00	limegreen 50,205,50 #32CD32	linen 250,240,230 #FAF0E6	magenta 255,0,255 #FF00FF	maroon 128,0,0 #800000
mediumaquamarine 102,205,170 #66CDAA	mediumblue 0,0,205 #0000CD	mediumorchid 186,85,211 #BA55D3	mediumpurple 147,112,219 #9370DB	mediumseagreen 60,179,113 #3CB371	mediumslateblue 123,104,238 #7B68EE	mediumspringgreen 0,250,154 #00FA9A	mediumturquoise 72,209,204 #48D1CC
mediumvioletred 199,21,133 #C71385	midnightblue 25,25,112 #191970	mintcream 245,255,250 #F5FFFA	mistyrose 55,228,225 #FFE4E1	moccasin 255,228,181 #FFE4B5	navajowhite 255,222,173 #FFDEAD	navy 0,0,128 #000080	oldlace 253,245,230 #FDF5E6
olive 128,128,0 #808000	olivedrab 107,142,35 #6B8E23	orange 255,165,0 #FFA500	orangered 255,69,0 #FF4500	orchid 218,112,214 #DA70D6	palegoldenrod 238,232,170 #EEE8AA	palegreen 152,251,152 #98FB98	paleturquoise 75,238,238 #AFEEEE
palevioletred 219,112,147 #DB7093	papayawhip 255,239,213 #FFEFD5	peachpuff 255,218,195 #FFDAB9	peru 205,133,63 #CD853F	pink 255,192,203 #FFC0CB	plum 221,160,221 #DDA0DD	powderblue 176,224,230 #B0E0E6	purple 128,0,128 #800080
rebeccapurple 102,51,153 #663399	red 255,0,0 #FF0000	rosybrown 188,143,143 #BC8F8F	royalblue 65,105,225 #4169E1	saddlebrown 139,69,19 #8B4513	salmon 250,128,114 #FA8072	sandybrown 44,164,96 #F4A460	seagreen 46,139,87 #2E8B57
seashell 155,245,238 #FFF5EE	sienna 160,82,45 #A0522D	silver 192,192,192 #C0C0C0	skyblue 135,206,235 #87CEEB	slateblue 106,90,205 #6A5ACD	slategray 112,128,144 #708090	snow 255,250,250 #FFFAFA	springgreen 0,255,127 #00FF7F
steelblue 70,130,180 #4682B4	tan 210,180,40 #D2B48C	teal 0,128,128 #008080	thistle 215,191,216 #D8BFD8	tomato 253,99,71 #FF6347	turquoise 64,224,208 #40E0D0	violet 238,130,238 #EE82EE	wheat 245,222,179 #F5DEB3
white 255,255,255 #FFFFFF	whitesmoke 245,245,245 #F5F5F5	yellow 255,255,0 #FFFF00	yellowgreen 154,205,50 #9ACD32				

FIGURE 13-2. The 140 color keywords in CSS (bear in mind that the colors look quite different on a screen). For a more accurate view, see *learningwebdesign.com/colornames.html*.

RGB Color

NOTE

I use the term "RGB" in this section because we are discussing the RGB color model used for specifying colors in CSS, but now you know that the colors you specify map to the gamut of the sRGB color space by default.

To recap, computers create the colors you see on a monitor by combining three colors of light: red, green, and blue. You can provide recipes (of sorts) for colors by telling the computer how much of each color to mix in. The amount of light in each color "channel" is typically described on a scale from 0 (none) to 255 (full blast), although the value can also be provided as a percent. Taken together, 255 colors in each channel can define around 16.7 million color combinations.

RGB colors in CSS are described by a series of three numbers: a red value, a green value, and a blue value, in that order. With the RGB color model, a pleasant lavender can be described as R:200, G:178, B:230. The closer the three values get to 255 (100%), the closer the resulting color gets to white (FIGURE 13-3). Wondering why the scale is from 0 to 255? See the **"Why 255?"** sidebar.

Why 255?

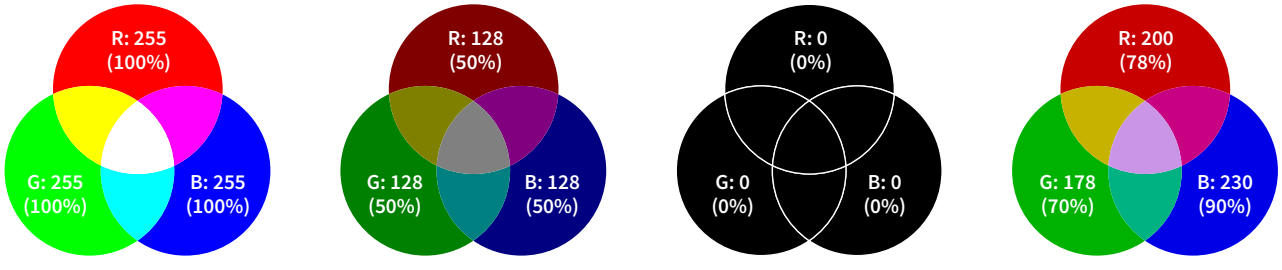
In RGB color spaces, 8 bits of information are devoted to each color channel. Because 8 bits can describe 256 shades ($2^8 = 256$), colors are measured on a scale from 0 to 255.

Picking a color

There are a number of ways to pick a color and find its RGB color values. One quick and easy option is to go to ***Google.com*** and search for "color picker"—*voilà*, a full-featured color picker (FIGURE 13-4, left)! If you have access to an image-editing program such as Figma or Adobe Photoshop, you can also use their built-in color pickers (FIGURE 13-4, center and right).

All of these color pickers show how the selected color would be expressed by a variety of color models, including RGB and HSL, which we'll look at shortly. (As mentioned previously, CMYK is used primarily for print media, so you won't use it except perhaps to translate print colors to their screen equivalents. The other color models in these color pickers are not currently appropriate for CSS.)

The RGB Color Model



RGB: 255, 255, 255
white

RGB: 128, 128, 128
gray

RGB: 0, 0, 0
black

RGB: 200, 178, 230
pleasant lavender

FIGURE 13-3. Computers create colors on a monitor by mixing different amounts of red, green, and blue light (thus, RGB). The color in the middle of each diagram shows what happens when the three color channels are combined. The more light there is in each channel (i.e., the higher the number value), the closer the combination is to white.

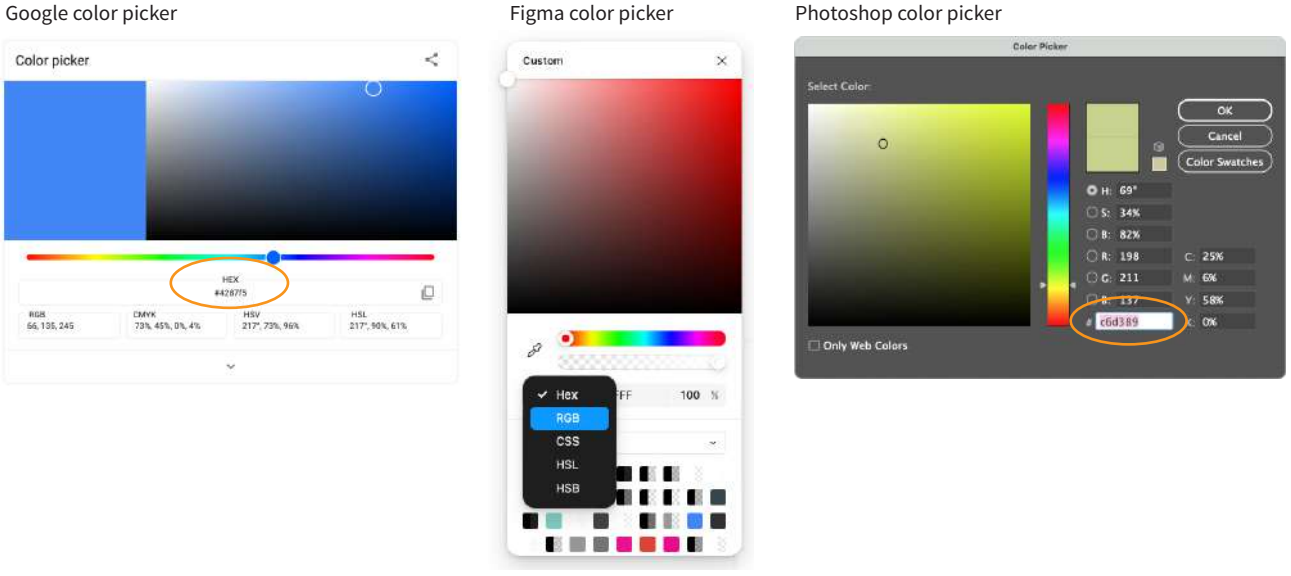


FIGURE 13-4. The Google, Figma, and Adobe Photoshop color pickers

When you select a color from the spectrum in the color picker, the red, green, and blue values are listed, as pointed out in FIGURE 13-4. If you look next to the # symbol, you'll find those same three values converted to hexadecimal equivalents so they are ready to go in a style sheet. I'll explain the six-digit hex values in a moment.

RGB functional notation

RGB color values may be specified with the **`rgb()`** **functional notation**, which is a syntax that provides the name of a function (**`rgb`** in this case), then passes it values within parentheses (three numeric values for red, green, and blue).

Going back to that pleasant lavender, you could add it to a style sheet by listing each value on a scale from 0 to 255, separated by character spaces (see **Note**):

```
background-color: rgb(200 178 230);
```

Or you could list them as percentage values, although that's less common:

```
background-color: rgb(78% 70% 90%);
```

You could also use the all-purpose **`color()`** function introduced in CSS Color Module Level 4. The first variable for the **`color()`** function indicates the color space, which is RGB in this example:

```
background-color: color(rgb 200 178 230);
```

As of this writing, however, browser support for **`color()`** is not as reliable as for **`rgb()`**, so check *CanIUse.com* for current support information, or better yet, stick with **`rgb()`** to be sure.

NOTE

Separating values with character spaces is a newer syntax, introduced in CSS Color Module Level 4. It's well supported by modern browsers, but if you need to support older browsers, use the legacy syntax that separates values with commas instead:

```
color: rgb(200, 178, 230);
```

Hexadecimal RGB values

Another (even more popular) method for specifying RGB colors is with a six-digit hexadecimal value, which you can see in the color pickers in FIGURE 13-4. These six digits represent the same three RGB values, except each has been converted into hexadecimal (or **hex** for short) equivalents. Note that hex RGB values are preceded by the # symbol and do not require the `rgb()` notation shown in the previous examples. They may be upper- or lowercase, but it is recommended that you be consistent:

```
color: #C8B2E6;
```

If your value happens to be made up of three pairs of double digits or letters, you can condense each pair down to one digit or letter, which slightly reduces the size of your file (however, full hex values are recommended):

`color: #FFCC00;` *is the same as* `color: #FC0;`

`color: #993366;` *is the same as* `color: #936;`

It's time to clarify what's going on with that six-digit string of characters. As I mentioned, the value is actually a series of three two-digit numbers, one each for red, green, and blue. But instead of decimal (base-10, the system we're used to), these values are written in hexadecimal, or base-16. FIGURE 13-5 shows the structure of the hex RGB value.

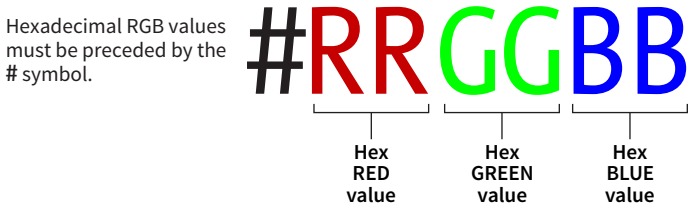


FIGURE 13-5. Hexadecimal RGB values are made up of three two-digit numbers, one for red, one for green, and one for blue.

The hexadecimal numbering system uses 16 digits: 0–9 and A–F (for representing the quantities 10–15). FIGURE 13-6 shows how this works. The hex system is used widely in computing because it reduces the space it takes to store certain information. For example, many RGB values are reduced from three to two digits once they're converted to hexadecimal.

Now that most image and web development software provides easy access to hexadecimal color values (as we saw in FIGURE 13-4), there isn't much need to translate RGB values to hex yourself, as we had to do back in the old days. Should you need to do this, there are plenty of decimal-to-hexadecimal converters online.

■ TIP

Handy Hex Values

White = #FFFFFF or #FFF
(the equivalent of 255 255 255)

Black = #000000 or #000
(the equivalent of 0 0 0)

■ AT A GLANCE

Specifying RGB Values

There are several options for providing RGB values in CSS. All of these examples specify white:

```
#FFFFFF
#FFF
rgb(255 255 255)
rgb(100% 100% 100%)
```

The Level 4 `color()` function is also an option, but check *CanIUse.com* for current browser support:

```
color(rgb 255 255 255)
color(rgb 100% 100% 100%)
```

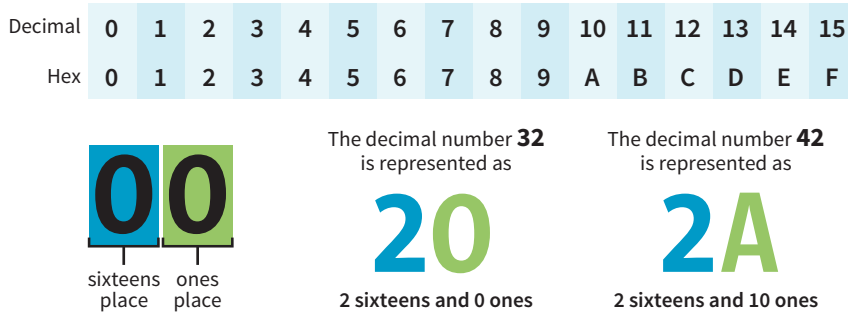

Decimal 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Hex 0 1 2 3 4 5 6 7 8 9 A B C D E F
00
sixteens place
ones place
The decimal number 32 is represented as
20
2 sixteens and 0 ones
The decimal number 42 is represented as
2A
2 sixteens and 10 ones

FIGURE 13-6. The hexadecimal numbering system is base-16.

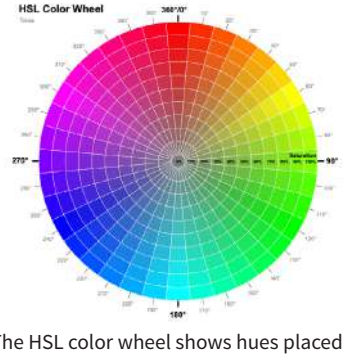
HSL Color

CSS Color Module Level 3 introduced the ability to specify colors by their HSL values: hue (color), saturation, and lightness or luminosity (see **Warning**).

In this model, the colors are spread out around a circle in the order of the rainbow, with red at the top (12 o'clock) position. Hue values are then measured in degrees around the circle: red at 0°/360°, green at 120°, and blue at 240°, with other colors in between (FIGURE 13-7). Saturation is a percentage value from 0% (gray) to 100% (color at full blast). Lightness (or brightness) is also a percentage value from 0% (darkest) to 100% (lightest). Despite using an alternative color model, HSL is just another way to define colors in the sRGB color space.

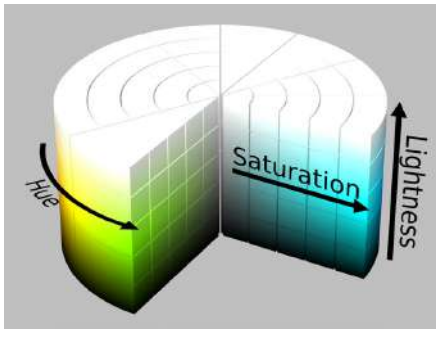
WARNING

HSL is not the same as HSB (hue, saturation, brightness), another color space provided in Photoshop and other image editors.



The HSL color wheel shows hues placed around a circle, with colors becoming more saturated toward the outer edges.

[Image credit: Websanity, *https://codepen.io/websanity/pen/QWNMRq*]



The HSL color model including the lightness axis

[Image credit: HSL and HSV. (2023, December 9). In Wikipedia. *https://en.wikipedia.org/wiki/HSL_and_HSV*. CC BY-SA 3.0 license]

FIGURE 13-7. The HSL color model

FIGURE 13-8 shows one hue, cyan (located at 180° on the wheel), with its associated saturation and lightness levels. You can see why developers find this system more intuitive to use. Once you lock into a hue, it is easy to make it

stronger, darker, or lighter by increasing or decreasing the percentage values. This makes it easier to handle shades of color programmatically as well. RGB values are not intuitive at all, although some practiced designers develop a feel for them.

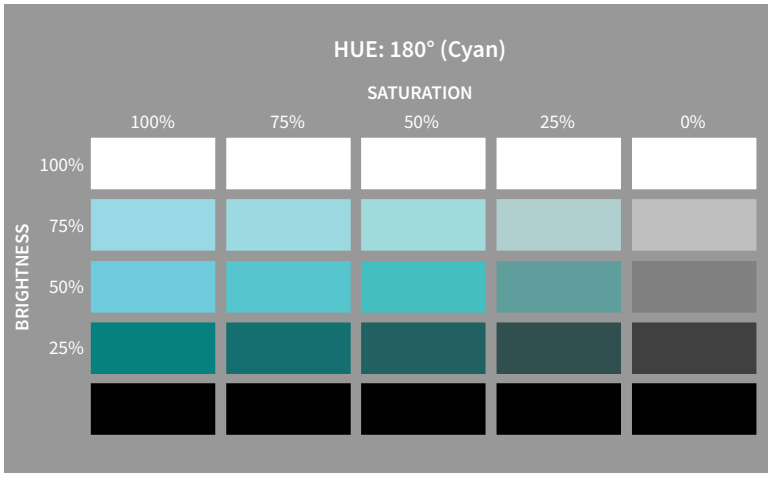


FIGURE 13-8. One hue in the HSL color model, with its associated saturation and lightness values

In the `hsl()` notation, HSL values are provided as a hue value and two percentages. They are never converted to hexadecimal values, as may be done for RGB. Here is that lavender from FIGURE 13-3 as it would be specified using HSL (see **Note**):

```
color: hsl(265 51% 80%);
```

The newer (but less well supported) Level 4 `color()` function may also be used for HSL values:

```
color: color(hsl 265 51% 80%);
```

NOTE

If you have a reason to support Internet Explorer or other old browsers, you should use the legacy syntax that separates the three values with commas, as demonstrated in the earlier RGB example.

Picking HSL color

There are a number of HSL color pickers online, including the Google color picker we looked at earlier. You can also give "A Most Excellent HSL Color Picker" (*hslpicker.com*) by Brandon Mathis a try.

OKLCH Color

Many developers have begun using the newer OKLCH color model introduced in CSS Color Module Level 4 to access brighter colors available on **high dynamic range** (**HDR**) monitors, such as Display P3 monitors. It is not as well supported as RGB and HSL, but the support landscape changes quickly, so it's definitely worth a quick introduction here.

Colors in the OKLCH model are specified with three values for perceived lightness (L), chroma (C), and hue (H), with an optional fourth value (a) for transparency:

Perceived lightness

Perceived lightness is expressed as a percentage from 0% to 100%. It is "perceived" lightness because it is based on how light a color appears to our eyes, unlike lightness in HSL, which is mathematically derived.

Chroma

Chroma is the vividness of the color, specified on a scale from 0.0 to 0.37. Colors with high chroma values have no white, black, or gray mixed in. Low chroma values produce colors that are "muddy" or less pure.

Hue

As in HSL, in OKLCH the colors of the rainbow are positioned around a wheel and specified in degrees (0deg to 360deg) or as numbers (0 to 360). However, colors do not occupy the same positions on the wheel, because they are based on human perception. In general, they have been shifted around 30 degrees. A few key positions are red: 30, yellow: 90, green: 140, cyan: 195, blue: 260, and magenta: 330.

Alpha transparency

If a fourth value is provided (preceded by a slash `/`), it describes how transparent the color should be in a range from 0% (fully opaque) to 100% (fully transparent). The transparency value may also be represented in a range from 0 to 1. I introduce transparency syntax in the next section.

OKLCH hasn't made its way into image editing tools quite yet, but you can choose and convert OKLCH color values at ***oklch.com*** (FIGURE 13-9). The pleasant lavender from the beginning of this chapter would be specified as:

```
oklch(79.96% 0.076 304.01)
```

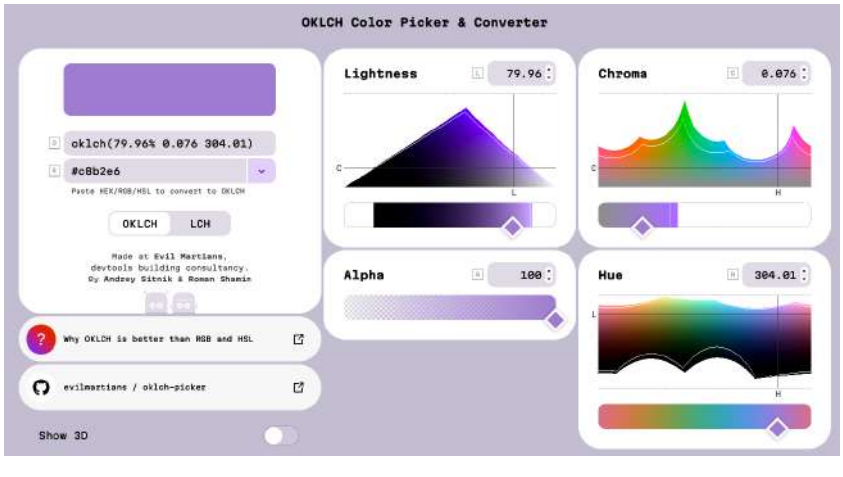


FIGURE 13-9. The OKLCH color picker (*oklch.com*)

Advanced Color Features

The sRGB color gamut has served us well, but displays are moving to wider gamuts such as the Display P3 color space. Recognizing that the syntax for specifying RGB and HSL colors in CSS Level 3 was going to be limiting, the CSS Working Group has been making huge strides in color specification in CSS Color Modules Levels 4 and 5. The details are certainly beyond the scope of this book, as color theory and systems are a complicated topic. But as usual, I want you to have a basic idea of what is possible, so I'll briefly go over some of the advanced color features here.

The primary change in the Level 4 module is that we are no longer restricted to the relatively small sRGB color gamut, thanks to new functions for specifying colors in alternative color spaces. Not all of them are appropriate for the web, but the authors of the Color Module wanted to be as flexible and future-facing as possible. In addition to `rgb()` and `hsl()`, the color functions in Level 4 are:

`hwb()`

For specifying colors in the sRGB color space using hue, whiteness, and blackness values.

`lab()` *and* `lch()`

For colors in the CIELAB color space, which is close to the gamut of colors the average human eye can see.

`oklab` *and* `oklch()`

Similar to CIELAB but with newer data. You can find an OKLCH color picker at *oklch.com*. OKLCH color is discussed further in the **"OKLCH Color"** section.

`color()`

An all-purpose color function that allows us to specify colors in a number of specialized color spaces. Values in the `color()` notation begin with a declaration of which color space is being used, followed by values appropriate to that color model. You've already seen `color(rgb # # #)` and `color(hsl # # #)`. To access some of the vivid colors available only in the Display P3 color space, you can use `color(display-p3 # # #)`. Be forewarned that as of this writing, support for `color()` is limited, so be sure to provide a backup color rule first that will get used on non-supporting browsers. Other color spaces mentioned for use with `color()` are Linear-light sRGB, A98 RGB (Adobe RGB), ProPhoto RGB (for photographs), CIE XYZ (similar to LAB), and Rec2020 (for the broadcast industry).

Level 5 introduces a few features that are experimental as of this writing, including the `color-mix()` function for mixing colors in a given color space, a "relative color" syntax for making modifications to an origin color, the `@color-profile` at-rule for naming an ICC (International Color Consortium) color profile for the document, and the `device-CMYK()` function for working with and converting to CMYK colors.

The following resources are a good starting point for learning more about color spaces and what the W3C is doing to expand color capabilities in CSS:

- CSS Color Module Level 4 (*w3.org/TR/css-color-4/*).
- CSS Color Module Level 5 (*w3.org/TR/css-color-5/*).
- "High Definition CSS Color Guide" on Chrome for Developers (*oreil.ly/pHsBE*).
- "Improving Color on the Web" by Dean Jackson (*oreil.ly/g1pdq*). This article is pretty old, but it does a really good job of making color gamuts understandable.

The OKLCH color model offers a number of advantages over RGB and HSL. It provides an intuitive lightness scale, similar to HSL, but allows us to specify super-bright colors not previously accessible in CSS. And because lightness is based on how the eye sees colors, hues with the same lightness value will appear to have the same level of lightness, which you can't count on with HSL. In addition, since you can control the chroma (you can think of it as the "purity" of the color), you don't get dull and grayish areas in gradients as you do with the RGB color model.

As of this writing, the major browsers have only recently added support for OKLCH, so you may want to provide a fallback color for non-supporting browsers:

```
h1 {
  color: #c8b2e6;
  color:  oklch(79.96% 0.076 304.01)
}
```

For a more in-depth introduction to the OKLCH color space, see the article "It's Time to Learn oklch Color" by Keith J. Grant (*oreil.ly/aTVLI*).

Adding Transparency

You can make any color as transparent or as opaque as you like by specifying an alpha transparency value after the color values, separated by a forward slash. The value controls an additional **alpha channel** that sets the level of transparency on a scale from 0 (fully transparent) to 1 (fully opaque). Percentage values may also be used. In this example, I've set the opacity of a black headline specified with RGB values to 50%; FIGURE 13-10 shows how this and a few other alpha settings might look in the browser:

```
h1 { color: rgb(0 0 0 / .5); }
```

FIGURE 13-10. A heading with various levels of transparency using alpha values

You can add alpha transparency values in this manner for any of the color functions in CSS Color Module Level 4 (see **Note**). Here is that same black headline using HSL and OKLCH notation, this time using a percentage for the alpha value:

```
h1 { color: hsl(0 0% 0% / 50%); }

h1 { color: oklch(0% 0 0 / 50%); }
```

NOTE

If you need to support older browsers, you'll need to use the specialized `rgba()` *and* `hsla()` *functions for adding transparency. In these legacy (but still valid) examples, the fourth value in the comma-separated list is the alpha value. OKLCH does not have a legacy syntax.*

```
rgba(0, 0, 0, .5)

hsla(0, 0%, 0%, .5)
```

Hexadecimal RGB color values can include a transparency level by tacking on the two-character alpha value at the end (**#RRGGBBAA**). If all four values contain double digits, the eight-character hex value with transparency can be abbreviated to four characters (**#RGBA**).

The alpha value is on a scale from 00 (totally transparent) to FF (totally opaque). This gets a bit tricky because converting percentages to hexadecimal is not exactly intuitive, but it can technically be done. Here's that same headline rule in hex RGB, where **80** specifies 50% transparency:

```
h1 {color: #00000080;}
```

Summing Up Color Values

It took us a quite a few pages to get here, but the process for picking and specifying colors in style sheets is actually easy. You can:

- Pick one of the predefined (but limited) color names.
- Use a color picker to select a color, copy down its RGB, HSL, or OKLCH values, and put those values in a style rule using the **rgb()**, **hsl()**, or **oklch()** notation. For RGB colors, you can use hex values for shorter code.

It's time to put those color values to work!

FOREGROUND COLOR

Now that you know how to write color values, let's get to the color-related properties. You can specify the foreground and background colors for any HTML element. There are **border-color** properties that take color values as well, but I'll get to those in **Chapter 15, Thinking Inside the Box**.

The foreground of an element consists of its text and border (if one is specified).

The **foreground** of an element consists of its text and border (if one is specified). You specify a foreground color with the **color** property, as you saw in the last chapter when we rolled it out to give text a little pizzazz. Here are the details for the **color** property one more time.

`color`

Values:	*color value (name or numeric)*
Default:	Depends on the browser and user's preferences
Applies to:	All elements
Inherits:	Yes

In the following example, the foreground of a **div** element is set to a green with the "olivedrab" color name. You can see that applying the **color** property to the **div** element means the color is inherited by the **p** and **b** elements it contains (FIGURE 13-11). The thick dashed border around the whole **div** is green

as well; however, if we were to apply a **border-color** property to this same element, that color would override the green foreground setting.

THE MARKUP

```
<div>
  <p>Welcome to <b>Black Goose Bistro</b>, where New American cuisine...
</div>
```

THE STYLE RULE

```
div {
  border: 8px dashed;
  color: olivedrab;
}
```

Welcome to Black Goose Bistro, where New American cuisine meets artisanal baked goods in a cozy, inviting atmosphere. Savor our chef's innovative takes on classic dishes, crafted from fresh, locally sourced ingredients. Whether you're indulging in a hearty brunch, a seasonal entrée, or a decadent pastry, every bite at Black Goose Bistro promises a blend of comfort and creativity.

FIGURE 13-11. Applying a color to the foreground of an element

BACKGROUND COLOR

Use **background-color** to apply a background color to any element.

background-color

Values: *color value (name or numeric)* | transparent

Default: transparent

Applies to: All elements

Inherits: No

A background color fills the **canvas** behind the element, which includes the content area and any padding (extra space) added around the content, extending behind the border out to its outer edge. Let's see what happens when we use the **background-color** property to make the background of the same sample **blockquote** light green (FIGURE 13-12):

```
div {
  border: 8px dashed;
  color: olivedrab;
  background-color: #b9d972; /* light green */
}
```

Welcome to Black Goose Bistro, where New American cuisine meets artisanal baked goods in a cozy, inviting atmosphere. Savor our chef's innovative takes on classic dishes, crafted from fresh, locally sourced ingredients. Whether you're indulging in a hearty brunch, a seasonal entrÃ©e, or a decadent pastry, every bite at Black Goose Bistro promises a blend of comfort and creativity.

FIGURE 13-12. Adding a light green background color to the sample `div`

NOTE

You can change the background painting area (the area on which fill colors are applied) with the `background-clip` *property. Its values are* `border-box`*, which draws the painting area to the outside edge of the border, as shown in* FIGURE 13-12*;* `padding-box`*, which starts the painting area on the outside edge of the padding area (the inside edge of the border); and* `content-box`*, which fills only the content area of the element.*

As expected, the background color fills the area behind the text, all the way to the border (see **Note**). Look closely at the gaps in the border, and you'll see that the background color goes to its outer edge. But that's where the background stops; if we apply a margin around this element, the background will not extend into the margin. We'll revisit all these components of an element when we talk about the CSS box model. For now, just know that, by default, if your border has gaps, the background will show through.

It's worth noting that background colors do not inherit, but because the default background setting for all elements is **`transparent`**, the parent's background color shows through its descendant elements. For example, you can change the background color of a whole page by applying the **`background-color`** property to the **`body`** element, and the color will show through all the elements on the page. The **`body`** element is an exception in that the background color doesn't get clipped to the content area, but instead extends to cover the entire viewport.

To color the background of the whole page, apply the background-color property to the body element.

You can change the background color of any element, including text-level elements that typically display inline. In this example, I've used the **`color`** and **`background-color`** properties to highlight defining terms marked up as **`dfn`** (definition element). You can see in FIGURE 13-13 that the background color fills the little box created by the inline **`dfn`** element.

THE MARKUP

```
<p><dfn>Sous vide</dfn> is a cooking method in which you submerge the
food into a water bath set to the target temperature.</p>
```

THE STYLE RULE

```
dfn {
  color: #darkolivegreen;
  background-color: yellow;
}
```

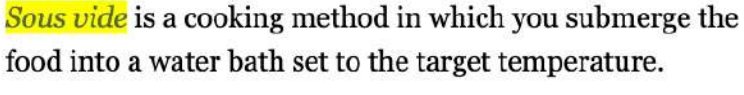


FIGURE 13-13. Applying the `color` and `background-color` properties to an inline element

There is one more colorful way to fill the background of an element—with **gradients** (colors that fade from one hue to another)—but I'm saving them for the end of this chapter.

PLAYING WITH OPACITY

Earlier, I talked about adding alpha transparency values to RGB, HSL, and OKLCH colors. There is another way to make an element slightly see-through—the **opacity** property.

opacity

Values: *number* (0 to 1)

Default: 1

Applies to: All elements

Inherits: No

The value for **opacity** is a number between 0 (completely transparent) and 1 (completely opaque). A value of .5 gives the element an opacity of 50%. The **opacity** setting applies to the entire element—both the foreground and the background (if one has been set). If you want to affect just one or the other, use a color value with alpha transparency instead.

The opacity setting applies to the entire element—both the foreground and the background.

In the following code example, a heading has been given a color of gold and a background color of white. When the **opacity** property is set, it allows the blue background of the page to show through both the text and the element box (FIGURE 13-14):

```
h1 {color: gold; background: white; opacity: .25;}

h1 {color: gold; background: white; opacity: .5;}

h1 {color: gold; background: white; opacity: 1;}
```

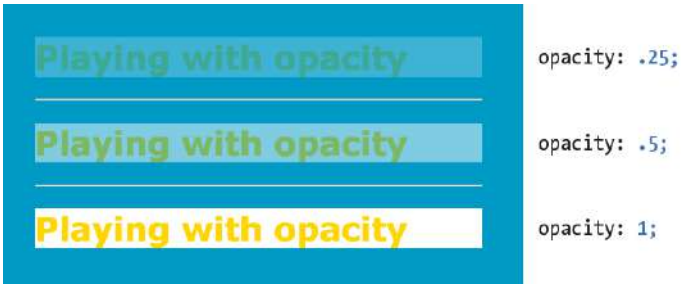


FIGURE 13-14. Setting the opacity on an element affects both the foreground and background colors.

It's time to try out foreground and background colors in EXERCISE 13-1 before moving on to background images.

■ DESIGN TIP

Limit the number of colors you use on a page. Nothing creates visual chaos faster than too many colors. Choose one dominant color and one highlight color. You may also use a couple of shades of each, but resist adding too many different hues.

EXERCISE 13-1. Adding color to a document

In this exercise, we'll add some new foreground and background colors to the Black Goose Bistro menu we've been working with. You should have enough experience writing style rules by this point that I won't hold your hand as much as I have in previous exercises. This time, you write the rules. You can check your work against the finished style sheet provided with the materials for this chapter (*learningwebdesign.com/6e/materials*).

1. Open the file *summermenu.html* in a text or code editor. It has the embedded style sheet that we created in the exercises in **Chapter 12, Formatting Text**. You got some experience with the `color` property in that chapter, so this time we'll try out some new value types.
2. Change the color of the `h1` to a shade of purple by using the RGB values R:153, G:51, B:153 with the `rgb()` function. Save the file and look at it in the browser. It just so happens that those values translate to 99, 33, and 99 in hexadecimal. Write the color value again, this time using hexadecimal notation (`#993399`). Save and refresh the page in your browser (it should be the same color). Feel free try out other RGB values.
3. Use the color picker of your choice and choose a color for the `h2` headings using HSL values. I'm choosing a dark purple with the value `hsl(299 65% 30%)`. Try increasing and decreasing the percentage values a few times to see how the color gets more or less saturated and more or less bright. This should give you a feel for how HSL values work.
4. Now let's add a light green color to the background of the entire page. Apply the `background-color` property to the `body` element and give it the value `#cdd7ae`. Of course, you can choose a different color of your own. (The color "tomato" from the previous exercise no longer works against the green, so I've deleted the style rules that use it. You can do the same, change them, or not worry about it.)
5. Next, we're going to make the background of the `header` white with 50% transparency (R:255, G:255, B:255, A:.5) so a hint of the background color shows through. Use the `rgb()` function and be sure to separate the color values from the alpha value with a forward slash. Save the file and take a look in the browser. (Note, if you are using Internet Explorer or another old browser, you'll need to use the legacy `rgba()` function with a comma-separated list.)
6. For fun, add an `opacity` declaration to the `header` and set it to .5. Save and reload to see the effect `opacity` has on the entire element. Once you've gotten the idea, you can delete this declaration.

When you're done, your page should something look like FIGURE 13-15. We'll be adding background images to this page later, so if you'd like to continue experimenting with different colors for different elements, make a copy of this document and give it a new name.

BLACK GOOSE BISTRO
· SUMMER MENU ·
Baker's Corner, Seekonk, Massachusetts
HOURS: MONDAY THROUGH THURSDAY: 11 to 9, FRIDAY AND SATURDAY: 11 to midnight
APPETIZERS
This season, we explore the spicy flavors of the southwest in our appetizer collection.
Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden. $3.95
Southwestern napoleons with lump crab — new item!
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas. $7.95
MAIN COURSES
Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.
Jerk rotisserie chicken with fried plantains — new item!
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. Very spicy. $12.95
Shrimp sate kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then
peanut sauce and jasmine rice. $12.95

purple
R:153, G:51, B:153
#993399 or #939

dark purple
hsl(299 65% 30%)

light green
R:205, G:215, B:174
#CDD7AE

FIGURE 13-15. The Black Goose Bistro menu page with background colors applied

Color Accessibility

Color contributes to both aesthetics and usability, so it is important to get it right. Be sure to choose combinations of colors for text and UI elements that have adequate contrast with the background so that people with visual disabilities such as color blindness can access all of the content on the page.

The Web Content Accessibility Guidelines (WCAG 2.0) define contrast as a measure of the difference in perceived "luminance" or brightness between two colors. This brightness difference is rated on a scale from 1:1 (white on white) to 21:1 (black on white). Text on a web page must have a minimum ratio of 4.5:1 (or 3:1 for large text).

Firefox, Chrome, and Edge all have ways of identifying color contrast issues in their built-in developer tools, even going as far as to show you where on the WCAG accessibility scale your current color combinations fall, and to help you to find more accessible combinations to use.

There are also some online tools that make it easy to calculate the brightness ratio, such as:

- WebAIM's Contrast Checker (*webaim.org/resources/contrastchecker/*)
- Accessible Web's Contrast Checker (*accessibleweb.com/color-contrast-checker/*), also available as a Chrome plug-in

To learn more about color accessibility, I recommend these resources:

- The article "Contrast and Color Accessibility" from WebAIM (*webaim.org/articles/contrast/*) is a great place to start.
- To see what any site looks like to users with various types of color blindness, enter the URL into the Colorblind Web Page Filter by Toptal at *toptal.com/designers/colorfilter*.

BACKGROUND IMAGES

We've talked about how to add images to the content of the document by using the **`img`** element, but most decorative images are added to pages and elements as backgrounds with CSS. After all, decorations such as tiling background patterns are firmly part of presentation, not structure. We've come a long way from the days when sites were giant graphics cut up and held together with tables (*shudder*).

In this section, we'll look at the collection of properties used to place and push around background images, starting with the basic **`background-image`** property.

Adding a Background Image

The **`background-image`** property adds a background image to any element. Its primary job is to provide the location of the image file.

`background-image`

Values: `url(`*location of image*`)` | `none`

Default: none

Applies to: All elements

Inherits: No

The value of **`background-image`** is a **`url()`** function that contains the location of the image.

■ AT A GLANCE

Background Properties

The properties related to the background are:

```
background-color
background-image
background-repeat
background-position
background-attachment
background-clip
background-size
background
```

The URL is relative to wherever the CSS rule is at the time. If the rule is in an embedded style sheet (a **style** element in the HTML document), then the pathname in the URL should be relative to the location of the HTML file. If the CSS rule is in an external style sheet, then the pathname to the image should be relative to the location of the *.css* file. This example backs up a level out of the directory where the *.css* file lives and finds the file *background.jpg* in the *images* directory:

```
background-image: url(../images/background.jpg);
```

As an alternative, providing site root relative URLs for images ensures that the background image can be found regardless of the location of the style rules. The root directory is indicated by a slash at the beginning of the URL:

```
background-image: url(/images/background.jpg);
```

The downside, as for all site root relative URLs, is that you won't be able to test it locally (from your own computer) unless you have it set up as a server or are using a web development build tool that generates a working root directory.

These examples and FIGURE 13-16 show background images applied behind a whole page (**body**) and a single **blockquote** element with padding and a border applied:

```
body {
  background-image: url(star.png);
}border


blockquote {
  background-image: url(dot.png);
  padding: 2em;
  border: 4px dashed;
}
```

In the figure, you can see the default behavior of **background-image**. The image starts in the top-left corner and tiles horizontally and vertically until the entire element is filled. Like background colors, tiling background images fill the area behind the content area, fill the extra padding space around the content, and extend to the outer edge of the border (if there is one). You can change the background painting area with the **background-clip** property.

If you provide both a **background-color** and a **background-image** to an element, the image is placed on top of the color. In fact, it is recommended that you *do* provide a backup color that is similar in hue, in case the background image fails to download.

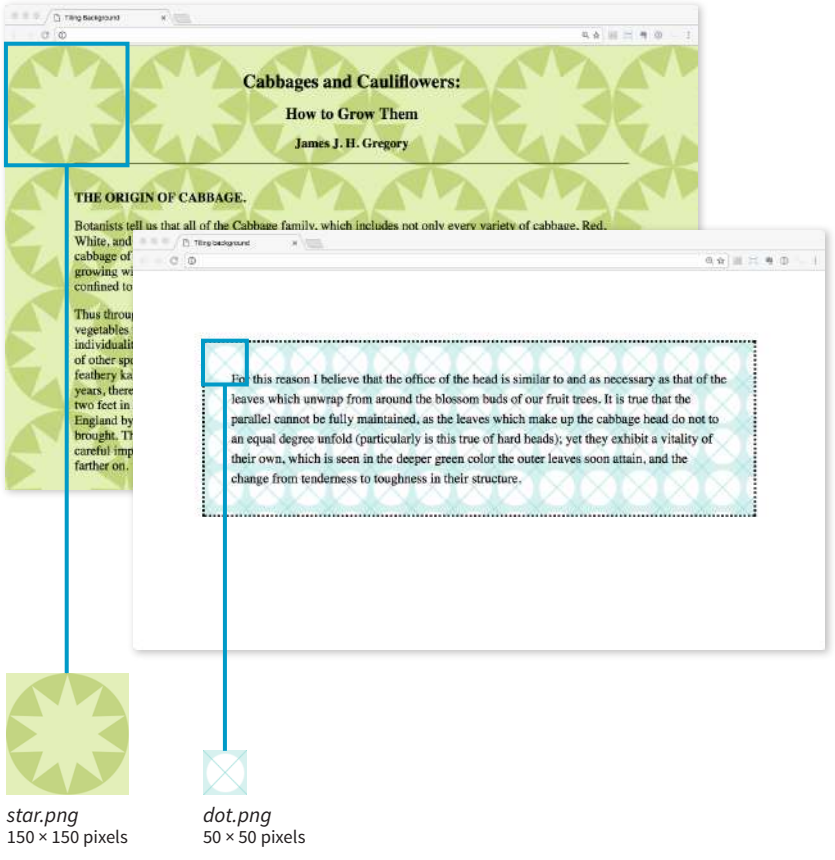

Cabbages and Cauliflowers:
How to Grow Them
James J. H. Gregory
THE ORIGIN OF CABBAGE.
For this reason I believe that the office of the head is similar to and as necessary as that of the leaves which unwrap from around the blossom buds of our fruit trees. It is true that the parallel cannot be fully maintained, as the leaves which make up the cabbage head do not to an equal degree unfold (particularly is this true of hard heads); yet they exhibit a vitality of their own, which is seen in the deeper green color the outer leaves soon attain, and the change from tenderness to toughness in their structure.
star.png
150 × 150 pixels
dot.png
50 × 50 pixels

FIGURE 13-16. Tiling background images added with the `background-image` property

Always specify a background color that is similar in hue to the background image, in case the file fails to load.

Additional Background Image Settings

If you don't care for the default behavior of tiled background images, fret not! There are plenty of properties for controlling the size, position, and direction of the repeating pattern. For the sake of saving space, I'm going to review them each quickly here to give you an overview, but I recommend you read the article "Tiling Background Images," available at ***learningwebdesign.com/articles***. There you will find detailed descriptions, examples, and exercises that allow you to try out each of the properties. You'll also learn how to add multiple background images to a single element and blend an image with the background color. If you've downloaded the book's supporting materials, you already have the required images and finished documents for the article's exercises in your *materials* folder.

That said, here are the types of things you can do with background images and the properties to do them. FIGURE 13-17 shows examples of each property in action. For the most part, the property name and the values they take are pretty intuitive:

■ DESIGN TIP

Tiling Background Images

When working with background images, keep these guidelines and tips in mind:

- Use a simple image that won't interfere with the legibility of the text over it.
- Always provide a `background-color` value that matches the primary color of the background image. If the background image fails to display, at least the overall design of the page will be similar. This is particularly important if the text color would be illegible against the browser's default white background.
- As usual for the web, keep the file size of background images as small as possible.

`background-repeat`

Controls how the image repeats. The default is **`repeat`**, but you can set it to not repeat (**`no-repeat`**), repeat only horizontally (**`repeat-x`**), or repeat only vertically (**`repeat-y`**). The **`space`** and **`round`** keyword values attempt to fill the available background painting area an even number of times by adding space or stretching the image, respectively.

`background-position`

Specifies the position of the **origin image** in the background. You can think of the origin image as the first image that is placed in the background, from which tiling images extend. The property can take a length measurement or a percentage that specifies an amount of offset from the top-left corner. It can also take the **`left`**, **`center`**, **`right`**, **`top`**, and **`bottom`** keywords, which specify the position relative to the outer edge of the element. Keywords are generally used in pairs.

`background-origin`

Specifies where the origin image should be placed relative to the element box. The values are **`border-box`**, **`padding-box`**, and **`content-box`**—terms that will make more sense once you've read **Chapter 15**.

`background-attachment`

Frees the background image from the content and allows it to stay fixed in one position while the rest of the content scrolls. The values are **`scroll`** (the default behavior), **`fixed`** (the image stays put relative to the browser window), and **`local`** (when the element has its own scrolling mechanism).

`background-size`

Allows you to change the size of the background image by setting its length, a percentage, or one of the keywords: **`auto`** (maintains proportions), **`cover`** (stretches to fill the canvas), or **`contain`** (sized just large enough to fill the width or height without changing the proportions of the image). Bear in mind that when resizing a bitmapped image like a PNG or JPEG larger, you run the risk that it will end up blurry.

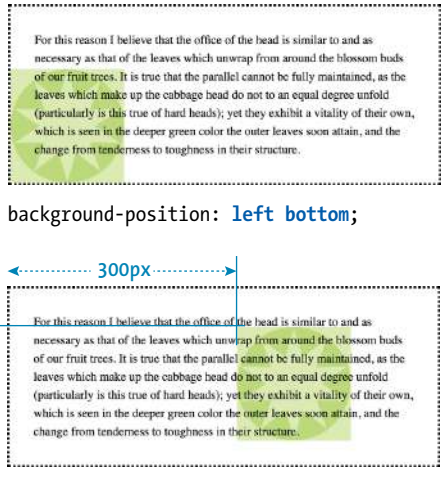
`background-blend-mode`

(Not shown in FIGURE 13-17.) Specifies how the background image should blend in with the background color and other background images applied to the element using layer interaction conventions found in image editing software such as Photoshop. Some values include **`normal`**, **`multiply`**, **`screen`**, **`overlay`**, **`darken`**, and **`difference`**.

Now you can try your hand at adding tiling background images in EXERCISE 13-2.

background-repeat

background-position

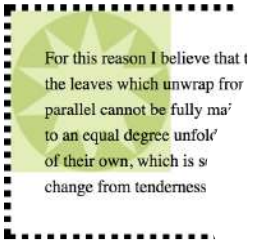

For this reason I believe that the office of the head is similar to and as necessary as that of the leaves which unwrap from around the blossom buds of our fruit trees. It is true that the parallel cannot be fully maintained, as the leaves which make up the cabbage head do not to an equal degree unfold (particularly is this true of hard heads); yet they exhibit a vitality of their own, which is seen in the deeper green color the outer leaves soon attain, and the change from tenderness to toughness in their structure.

```
background-position: left bottom;
```

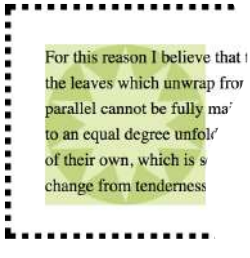

300px
50px
For this reason I believe that the office of the head is similar to and as necessary as that of the leaves which unwrap from around the blossom buds of our fruit trees. It is true that the parallel cannot be fully maintained, as the leaves which make up the cabbage head do not to an equal degree unfold (particularly is this true of hard heads); yet they exhibit a vitality of their own, which is seen in the deeper green color the outer leaves soon attain, and the change from tenderness to toughness in their structure.

```
background-position: 300px 50px;
```

background-origin


For this reason I believe that t
the leaves which unwrap fro
parallel cannot be fully ma
to an equal degree unfol
of their own, which is s
change from tenderness

```
background-origin: padding-box;
```


For this reason I believe that
the leaves which unwrap fro
parallel cannot be fully ma
to an equal degree unfol
of their own, which is s
change from tenderness

```
background-origin: content-box;
```

background-attachment

```
background-attachment: fixed;
```

The document has a large, nonrepeating image. When `background-attachment` is set to `fixed`, the image stays in its position relative to the browser viewing area and does not scroll with the content.

background-size

target.png
300 × 300 pixels

```
background-size: 600px 300px;
```

```
background-size: cover;
```

```
background-size: contain;
```

FIGURE 13-17. Examples of background image fine-tuning properties

EXERCISE 13-2. Adding a few background images

In this exercise, we're going to add a simple tiling background image to the menu.

1. Add a declaration to the **body** style rule that makes the image *bullseye.png* tile in the background of the page. Be sure to include the pathname relative to the style sheet (in this case, the current HTML document). The image is in the *images* directory:

```
body {
  ...
  background-image: url(images/bullseye.png);
;
```

   Easy, isn't it? When you save and view the page in the browser, you should see the circles in the background, as in FIGURE 13-18. Resize your browser window and notice that the background tile always starts from the top-left corner.

   I want to point out that *bullseye.png* is a slightly transparent PNG graphic, so it blends into any background color. Try temporarily changing the **background-color** for the **body** element by adding a second **background-color** declaration lower in the stack, so it overrides the previous one. Play around with different colors and notice how the circles blend in.

2. I'd like the pattern to stay centered at the top of the viewport so it aligns with the headline. Use the **background-position** property to set the horizontal and vertical position of the first tile:

```
  ...
  background-image: url(images/bullseye.png);
  background-position: top center;
}
```

   Save the file, refresh the browser, and make the window narrower and wider, noticing the new relationship of the pattern with the heading (FIGURE 13-18).

BLACK GOOSE BISTRO
· SUMMER MENU ·
APPETIZERS
This season, we explore the spicy flavors of the southwest in our appetizer collection.
Black bean purses
Spicy black bean and a blend of mexican cheeses wrapped in sheets of phyllo and baked until golden.
Southwestern napoleons with lump crab — new item!
Layers of light lump crab meat, bean and corn salsa, and our handmade flour tortillas.
MAIN COURSES
Big, bold flavors are the name of the game this summer. Allow us to assist you with finding the perfect wine.
Jerk rotisserie chicken with fried plantains — new item!
Tender chicken slow-roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and fresh mango. Very spicy.
Shrimp satè kebabs with peanut sauce
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection. Served with spicy peanut sauce and jasmine rice.

FIGURE 13-18. A tiled background pattern for the menu

3. The next thing we'll do with the main tiled background is make it stay in a fixed position when the page scrolls. Use the **background-attachment** property for this, setting it to **fixed** (the default is **scroll**):

```
  ...
  background-image: url(images/bullseye.png);
  background-position: top center;
  background-attachment: fixed;
}
```

   Save the file, reload the page, and try scrolling it. The background pattern shouldn't move. This is a useful technique for using big hero images in the background of a page, which is a pretty common sight these days.

4. Now let's play around with some background image properties in the **header**. In the style rule for the **header** element, add the *purpledot.png* image, only this time, restrict the tiling to only the horizontal direction:

```
header {
  ...
  background-image: url(purpledot.png);
  background-repeat: repeat-x;
}
```

5. You should have a row of dots across the top of the **header** element only, as shown in FIGURE 13-19 (top). Now, delete those declarations (or comment them out to save it for later), and let's try something new. This time we'll add a single, centered black goose icon (*blackgoose.png*) that is as large as it can be yet still fits in the header, using the **contain** value of the **background-size** property. Add the following declarations, save the document, and refresh the browser (FIGURE 13-19, bottom):

```
header {
  ...
  background-image: url(images/blackgoose.png);
  background-repeat: no-repeat;
  background-position: center;
  background-size: contain;
}
```

6. In this last step, you'll learn how to put multiple background images in the same element. We'll be adding a little goose silhouette illustration to the **header** along with the existing icon (FIGURE 13-20). To apply multiple background images to an element, provide the values in a comma-separated list. Start by adding a second image (*gooseshadow.png*) to the existing **background-image** declaration (don't forget the comma). Images earlier in the list will be drawn on top of images later in the list.

EXERCISE 13-2. Continued

```
background-image: url(images/purpledot.png);
background-repeat: repeat-x;
```

```
background-image: url(images/blackgoose.png);
background-repeat: no-repeat;
background-position: center;
background-size: contain;
```

FIGURE 13-19. Controlling the repeat and size of background images in the header of the page

```
header { ...
  background-image:
    url(images/gooseshadow.png),
    url(images/blackgoose.png);
```

Additional background-related properties also go in comma-separated lists; the first value applies to the first image, the second value to the second, and so on. You need to be careful to get the order right.

```
  background-repeat: no-repeat, no-repeat;
  background-position: 3em bottom, center;
  background-size: auto, contain;
}
```

And we're done! I encourage you to play around with the background properties and see what the other values do. It's the best way to get a feel for them.

FIGURE 13-20. The header with a small flock of geese in the background

The Shorthand background Property

The style rules for setting colors and background images with tiling adjustments can get quite verbose. But thankfully, you can use the handy **background** property to specify *all* of your background styles in one tidy declaration.

background

Values: *background-color* *background-image* *background-repeat* *background-attachment* *background-position* *background-clip* *background-origin* *background-size*

Default: See individual properties

Applies to: All elements

Inherits: No

The value of the **background** property lets you put all the values you want to apply with individual **background-** properties into one declaration. It takes values for both background color and images. For example, this one background rule:

```
body { background: white url(star.png) no-repeat right top fixed; }
```

replaces this rule with five separate declarations:

```
body {
  background-color: white;
  background-image: url(star.png);
  background-repeat: no-repeat;
  background-position: right top;
  background-attachment: fixed;
}
```

All of the property values for **background** are optional, and they may appear in any order. The only restriction is that when you are providing the coordinates for the **background-position** property, the horizontal value must appear first, immediately followed by the vertical value. As with any shorthand property, be aware that if any value is omitted, it will be reset to its default value. See the **"Watch Out for Overrides"** sidebar.

Before we move on to gradient background fills, there is another technique I want you to know about even though it has to do with inline images (placed with the **img** element), not background images. You can use the **object-fit** and **object-position** properties to make an image fill its container, much in the way we used **background-size** to fill the background. The **"Resizing Inline Images with object-fit and object-position"** sidebar has all the details.

Watch Out for Overrides

The **background** property is efficient, but use it carefully. I've addressed this before, but it bears repeating. Because **background** is a shorthand property, when you omit a value, that property will be reset to its default. Be careful that you do not accidentally override style rules earlier in the style sheet with a later shorthand rule that reverts your settings to their defaults.

In this example, the background image *dots.gif* will *not* be applied to **h3** elements, because by omitting the value for **background-image**, you essentially set that value to **none**:

```
h1, h2, h3 {
  background: red url(dots.gif) repeat-x;
}
h3 {
  background: green;
}
```

To override particular properties, use the specific background property you intend to change. For example, if the intent in the preceding example were to change just the background color of **h3** elements, the **background-color** property would be the correct choice.

Resizing Inline Images with object-fit and object-position

object-fit

Values: contain | cover | fill | none | scale-down

Default: fill

Applies to: Replaced elements

Inherits: No

object-position

Values: *two space-separated values* (top | bottom | left | right | center | percentage values | lengths)

Default: 50% 50%

Applies to: Replaced elements

Inherits: Yes

The **background-size** property allows us to size a background image so that it keeps its original proportions and covers the whole element (**cover**) or is completely contained without getting cropped (**contain**). You can see these values at work in FIGURE 13-17.

You can get that same effect for inline images placed with the **img** element by using the **object-fit** property and fine-tune their position with **object-position**. These properties are useful for when you need an image to fill a container in the layout but not get distorted in the process. In the past, we would have needed to open the image in an editing tool and make a version that filled the container. Now, we can allow the browser to do the cropping and resizing as necessary, and even make adjustments on the fly if the container changes size and proportions in a responsive layout.

In this example, I have an image that is 450 by 600 pixels in portrait orientation. I want it to fit into a container that is 600 x 400px in landscape orientation, but I don't want it to be squished or stretched. The markup shows the image inside its container. In the styles, I've specified the container's width and height. I've also given the image a width and height of 100%, as the first step in making it fill its container; however, if I'd stopped there, the image would be distorted (FIGURE 13-21, **4**).

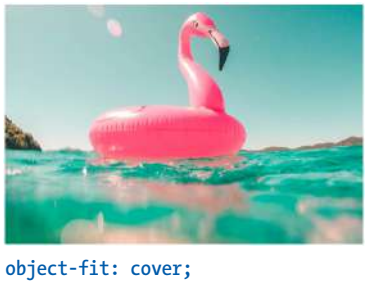
Continued...

When there are no styles applied to the **img** element, it appears at actual size, and in this example, hangs out of its container element.

1

```
object-fit: contain;
```

2



```
object-fit: cover;
```

3

```
object-fit: cover;
object-position: center -125px;
```

4

```
object-fit: fill;
```

same as

```
img { width: 100%; height: 100%; }
```

FIGURE 13-21. *The* **object-fit** *and* **object-position** *properties (photo by Vicko Mozara on Unsplash)*

Resizing Inline Images, continued

THE MARKUP

```
<div class="container">
  <img src="flamingo.jpg" alt="photograph of flamingo pool toy">
</div>
```

THE STYLES

```
.container {
  width: 600px;
  height: 400px;
  background-color: lightcoral;
}
img {
  width: 100%;
  height: 100%;
}
```

To fit the entire image in the container while retaining its proportions, we can use the **object-fit** property with the **contain** value (FIGURE 13-21, **1**)—note that the image is not cropped, but the colored container background shows around the image:

```
img {
  width: 100%;
  height: 100%;
  object-fit: contain;
}
```

If we want the entire container to be filled without distorting the image, we can use the **cover** value for **object-fit** (FIGURE 13-21, **2**). In the example, the top and bottom edges of the image are cropped out of view:

```
img {
  width: 100%;
  height: 100%;
  object-fit: cover;
}
```

By default, the image is centered vertically and horizontally in its container, but we can change that with the **object-position** property. To change the position, use a pair of keyword values (**top**, **bottom**, **left**, **right**, **center**), percentage values, or lengths. The first value provides the horizontal position, and the second value indicates the vertical placement. In this example, the image stays centered horizontally in the container but I've indicated that I want it moved *up* by 125 pixels by specifying a negative vertical value (FIGURE 13-21, **3**):

```
img {
  width: 100%;
  height: 100%;
  object-fit: cover;
  object-position: center -125px;
}
```

The other values for **object-fit** are **fill** (fills the box by distorting the image; FIGURE 13-21, 4), **none** (the image is not resized), and **scale-down** (makes the image smaller to fit, similar to **contain**, but does not resize the image larger). I've been focusing on images in this explanation, but you can use **object-fit** and **object-position** with any replaced element (such as videos).

LIKE A RAINBOW (GRADIENTS)

A **gradient** is a transition from one color to another, sometimes through multiple colors. When you specify color gradients using CSS, the browser handles the task of rendering the color blends. But be aware, although they are specified with code, gradients are *images*. They just happen to be generated on the fly. A gradient image has no intrinsic size or proportions; the size matches the element it gets applied to. Gradients can be applied anywhere an image may be applied: `background-image`, `border-image`, and `list-style-image`. I'll stick with `background-image` examples in this chapter.

Gradients are images that browsers generate on the fly. Use them as you would use a background image.

There are three types of gradients:

- **Linear gradients** change colors along a line, from one edge of the element to the other.
- **Radial gradients** start at a point and spread outward in a circular or elliptical shape.
- **Conic gradients** rotate around a center point. I will not be covering conic gradients in this book, but you can learn more about them on MDN Web Docs (*oreil.ly/49cmu*).

Linear Gradients

The `linear-gradient()` notation provides the angle of the **gradient line** and one or more points along that line where the pure color is positioned (**color stops**). You can use color names or any of the numerical color values discussed earlier in the chapter, including transparency. The angle of the gradient line is specified in degrees (*n*`deg`) or with keywords. With degrees, `0deg` points upward, and positive angles go around clockwise, so that `90deg` points to the right. Therefore, if you want to go from aqua on the top edge to green on the bottom edge, set the rotation to `180deg`:

```
background-image: linear-gradient(180deg, aqua, green);
```

The keywords describe direction in increments of 90° (`to top`, `to right`, `to bottom`, `to left`). So, our `180deg` gradient could also be specified with the `to bottom` keyword. The result is shown in FIGURE 13-22 (top):

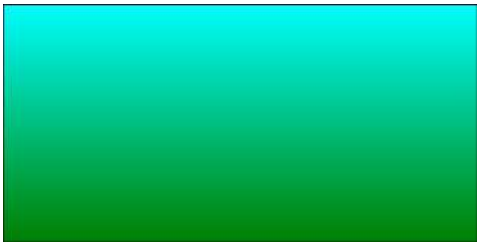
```
background-image: linear-gradient(to bottom, aqua, green);
```

You can use the "to" syntax to point to corners as well. The following gradient would be drawn from the bottom-left corner to the top-right corner. The resulting angle of a gradient drawn between corners is determined by the aspect ratio of the box:

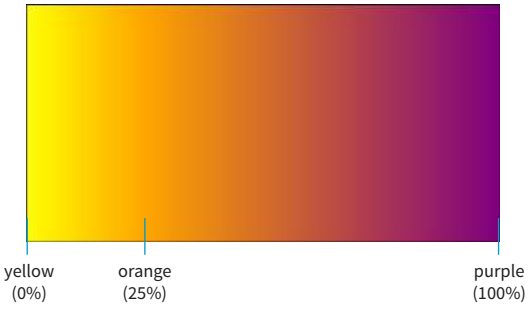
```
background-image: linear-gradient(to top right, aqua, green);
```

In the following example, the gradient goes from left to right (`90deg`) and includes a third color, orange, which appears 25% of the way across the gradient line (FIGURE 13-22, middle). You can see that the placement of the color

```
linear-gradient(180deg, aqua, green);
or
linear-gradient(to bottom, aqua, green);
```



```
linear-gradient(90deg, yellow, orange 25%, purple);
```




```
linear-gradient(54deg, red, orange, yellow, green, blue 50%);
```

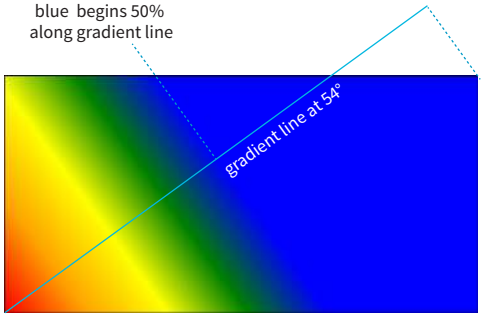


FIGURE 13-22. Examples of linear gradients

PERFORMANCE TIP

Gradients offer both advantages and disadvantages when it comes to performance. On the plus side, they do not require an extra call to the server and require fewer bytes to download than images. On the other hand, all that rendering on the fly requires time and processing power, which can hurt performance. If the gradients cover a small area, or are not particularly complex, this most likely won't be an issue. But for gradients that cover a larger area or have more complexity to them (radial gradients are the worst culprits), the performance impact can be problematic, particularly on mobile devices where processing power may be limited. Consider serving a separate style sheet without complex gradients to mobile devices.

stop is indicated after the color value. You can use percentages or any length measurement. The first and last color stops don't require positions because they are set to 0% and 100%, respectively, by default:

```
background-image: linear-gradient(90deg, yellow, orange 25%, purple);
```

You certainly aren't limited to right angles. Specify any degree you like to make the linear gradient head in that direction. You can also specify as many colors as you like. If no positions are specified, the colors are spaced evenly across the length of the gradient line. If you position the last color stop short

of the end of the gradient line (such as the blue at 50% in this example), the last color continues to the end of the gradient line (FIGURE 13-22, bottom):

```
background-image: linear-gradient(54deg, red, orange, yellow, green,
blue 50%);
```

These examples are pretty garish, but if you choose your colors and stops right, gradients are a nice way to give elements subtle shading and a 3D appearance. The button in FIGURE 13-23 uses a background gradient to achieve a 3D look without images.

FIGURE 13-23. A 3D button made with only CSS

```
a.button-like {
  background: linear-gradient(to bottom, #e2e2e2 0%, #dbdbdb 50%,
  #d1d1d1 51%, #fefefe 100%);
}
```

That concludes our quick-and-dirty tour of linear gradients. You should know that I really only scratched the surface of linear gradient behavior and possibilities, so you may want to check out the resources in the **"Further Reading"** sidebar. Now, it's time to move on to radial gradients.

FURTHER READING

The most in-depth coverage of CSS gradient syntax that I've read is in Eric Meyer's book *Colors, Backgrounds, and Gradients* (O'Reilly). The same content is available in *CSS: The Definitive Guide*, by Eric Meyer and Estelle Weyl (also from O'Reilly).

Online, I recommend these overviews and tutorials:

- "CSS Gradients" by Chris Coyierk (*css-tricks.com/css3-gradients/*)
- "Using CSS Gradients" at MDN Web Docs (*developer.mozilla.org/en-US/docs/Web/CSS/CSS_Images/Using_CSS_gradients*)
- "CSS3 Gradients," part of the *CSS Mine* ebook by Martin Michalek (*cssmine.com/ebook/css3-gradients*)

Radial Gradients

Radial gradients, as the name suggests, radiate out from a point in a circle along a **gradient ray** (like a gradient line, but always pointing outward from the center). At minimum, a radial gradient requires two color stops, as shown in this example:

```
background-image: radial-gradient(yellow, green);
```

By default, the gradient fills the available background area, and its center is positioned in the center of the element (FIGURE 13-24). The result is an ellipse if the containing element is a rectangle and a circle if the element is square.

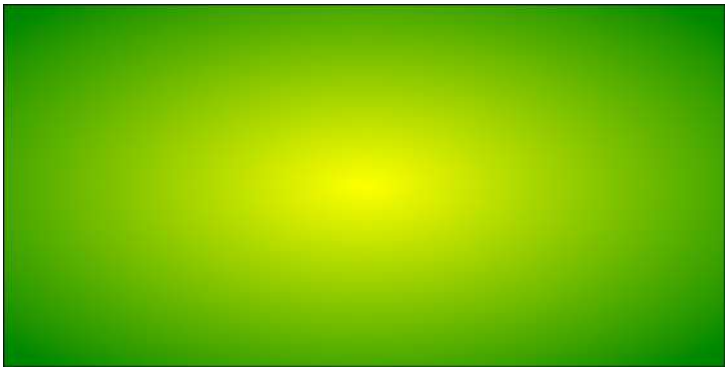
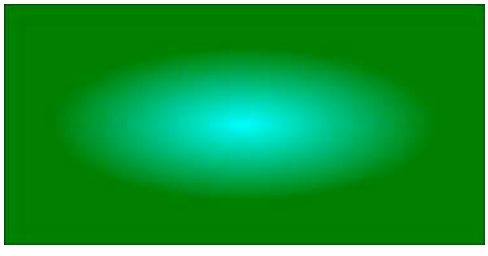


FIGURE 13-24. A minimal radial gradient with default size and position

That looks pretty spiffy already, but you don't have to settle for the default. The `radial-gradient()` notation allows you to specify the shape, size, and center position of the gradient:

```
radial-gradient(circle, yellow, green);
```

```
radial-gradient(200px 80px, aqua, green);
```



```
radial-gradient(farthest-side at right bottom, yellow, orange 50%, purple);
```

FIGURE 13-25. Examples of sizing and positioning radial gradients

Shape

In most cases, the shape of the radial gradient will result from the shape of the element or an explicit size you apply to it, but you can also specify the shape by using the **`circle`** or **`ellipse`** keywords. When you make a gradient a **`circle`** (without conflicting size specifications), it stays circular even when it is in a rectangular element (FIGURE 13-25, top):

```
background-image: radial-gradient(circle, yellow, green);
```

Size

The size of the radial gradient can be specified in length units or percentages, which apply to the gradient ray, or with keywords. If you supply just one length, it is used for both width and height, resulting in a circle. When you provide two lengths, the first one is the horizontal measurement and the second is vertical (FIGURE 13-25, middle). For ellipses, you can provide percentage values as well, or mix percentages with length values:

```
background-image: radial-gradient(200px 80px, aqua, green);
```

There are also four keywords—**closest-side**, **closest-corner**, **farthest-side**, and **farthest-corner**—that set the length of the gradient ray relative to points in the containing element.

Position

By default, the center of the gradient is positioned at **center center**, but you can change that by using the positioning syntax I covered for the **background-position** property in **"Additional Background Image Settings."** The syntax is the same, but it should be preceded by the **at** keyword, as in this example (FIGURE 13-25, bottom). Notice that in this example, I have included an additional color stop of orange at the 50% mark:

```
background-image: radial-gradient(farthest-side at right bottom,
yellow, orange 50%, purple);
```

Designing Gradients

Although it is not impossible to write the code for gradients by hand, an online gradient tool makes the job much easier! One option is the Ultimate CSS Gradient Generator from Colorzilla (*colorzilla.com/gradient-editor/*), shown in FIGURE 13-26. Newer gradient generators, such as the ones at *gradient.style* and *joshwcomeau.com/gradient-generator*, include additional color models, such as OKLCH. Simply enter as many color stops as you'd like, slide the sliders around until you get the look you want, and then copy the code.

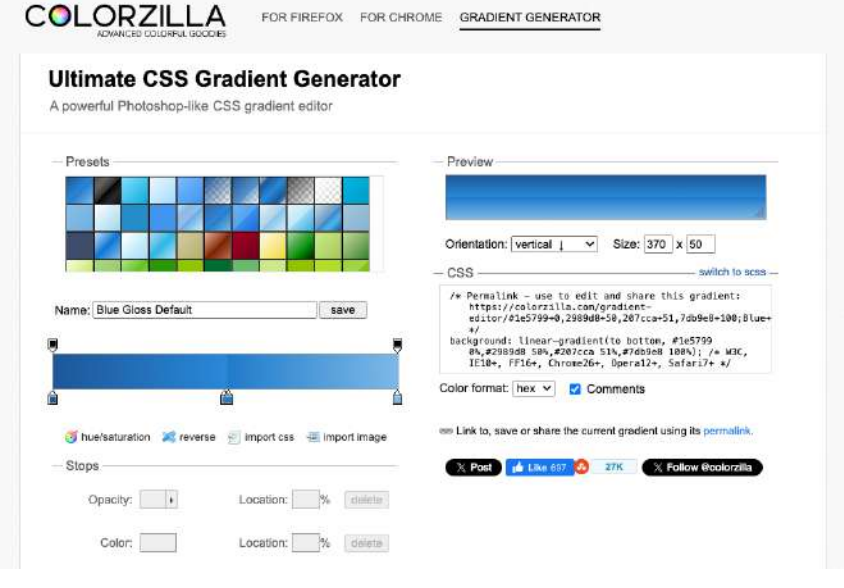


FIGURE 13-26. The Ultimate CSS Gradient Generator (*www.colorzilla.com/gradient-editor*) makes creating CSS gradients a breeze.

FIGURE 13-27. The CSS3 Patterns Gallery, assembled by Lea Verou (*lea.verou.me/css3patterns*). You may also enjoy Lea's book, *CSS Secrets: Better Solutions to Everyday Web Design Problems* (O'Reilly).

If you want your mind blown, take a look at the wild background patterns made with gradients assembled by Lea Verou in her CSS3 Patterns Gallery (*lea.verou.me/css3patterns*; FIGURE 13-27). It's inspirational, and you can take a peek at the code used to create them. CSS Background Patterns from MagicPattern is a similar site that allows you to copy the code and use it in your own designs (*magicpattern.design/tools/css-backgrounds*).

WRAPPING IT UP

We've covered a lot of ground (and *background*!) in this chapter. After getting to know how computers handle color, we looked at ways to set the foreground and background colors for an element by using various numeric systems and color names. We looked at options for adjusting the level of transparency, with color notations and with the **opacity** property. We also added tiling background images to the page and to elements. And finally, we took a look at how linear and radial gradients can be used as background images.

TEST YOURSELF

It's time to test your color and background prowess.

1. Match the following rules with their respective samples, shown in FIGURE 13-28. All of the samples in the figure use the same source document, consisting of one paragraph element to which some padding and a border have been applied. We touched on some of the properties only briefly in this chapter, but they should be fairly intuitive. If you'd like more explanation, read the "Tiling Background Images" article on *learningwebdesign.com/articles*.

   a.
   ```
   body {
     background-image: url(graphic.gif);
   }
   ```

   b.
   ```
   p {
     background-image: url(graphic.gif);
     background-repeat: no-repeat;
     background-position: 50% 0%;
   }
   ```

   c.
   ```
   body {
     background-image: url(graphic.gif);
     background-repeat: repeat-x;
   }
   ```

   d.
   ```
   p {
     background: url(graphic.gif) no-repeat right center;
   }
   ```

e.
```
body {
  background-image: url(graphic.gif);
  background-repeat: repeat-y;
  }
```

f.
```
body {
  background: url(graphic.gif) no-repeat right center;
}
```

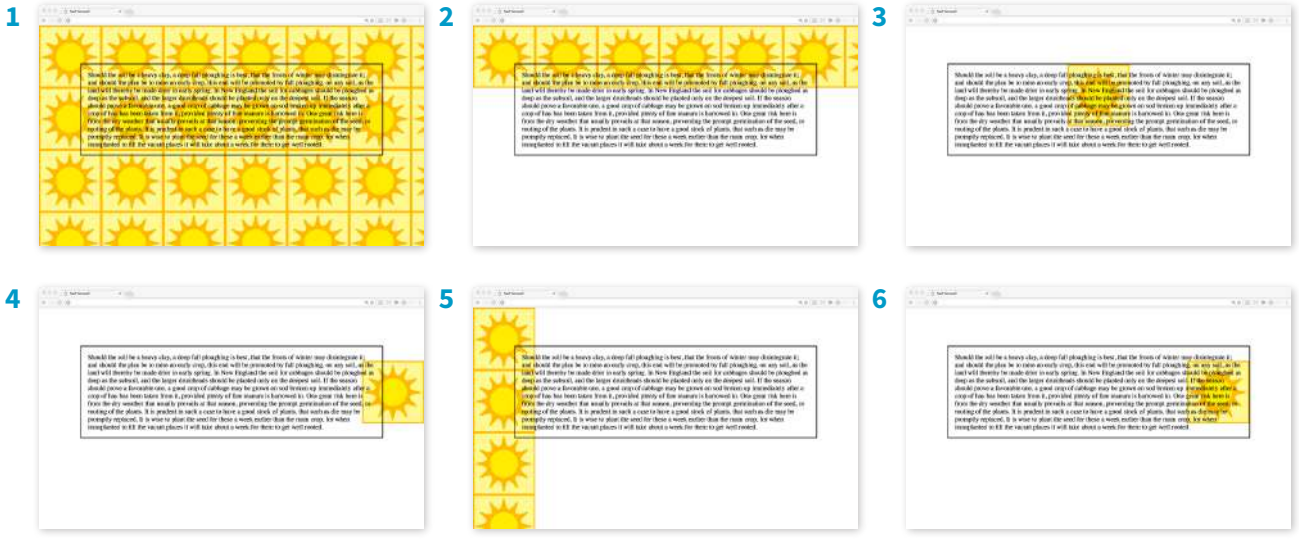


FIGURE 13-28. Samples for question 1

2. A system for specifying colors numerically is called a _____.
3. The range of colors a device or color space can represent is called its _____.
4. Any system, analog or digital, that provides a method for accurately communicating and reproducing colors is called a _____.
5. Which of these areas gets filled with a background color by default?
   a. The area behind the content
   b. Any padding added around the content
   c. The area under the border
   d. The margin space around the element
   e. All of the above
   f. a and b
   g. a, b, and c

6. Which of these is *not* a way to specify the color white in CSS?

   a. `#ffffff`

   b. `#FFF`

   c. `rgb(255 255 255)`

   d. `rgb(FF, FF, FF)`

   e. `white`

   f. `rgb(100%, 100%, 100%)`

CSS REVIEW: COLOR AND BACKGROUND PROPERTIES

Here is a summary of the properties covered in this chapter, in alphabetical order.

Property	Description
`background`	Shorthand property that combines background properties
`background-attachment`	Specifies whether the background image scrolls or is fixed
`background-clip`	Specifies how far the background image should extend
`background-color`	Specifies the background color for an element
`background-image`	Provides the location of an image to use as a background
`background-origin`	Determines how the **`background-position`** is calculated (from edge of border, padding, or content box)
`background-position`	Specifies the location of the origin background image
`background-repeat`	Specifies whether and how a background image repeats (tiles)
`background-size`	Specifies the size of the background image
`color`	Specifies the foreground (text and border) color
`opacity`	Specifies the transparency level of the foreground and background
For replaced elements (such as the img element) in their containers	
`object-fit`	Specifies how the content of a replaced element should be resized to fit its container
`object-position`	Specifies the alignment of the replaced element's content within the element's box

CHAPTER
14

SELECTORS

PLUS EXTERNAL STYLE SHEETS

IN THIS CHAPTER

Selector review

Attribute selectors

Pseudo-class selectors

Pseudo-element selectors

Linking to an external style sheet

Importing an external style sheet

Knowing how to use selectors strategically and efficiently to target content is a core web development skill. So far, the majority of the selectors we've been using call out target elements by name, whether it's one element type in a document, a group of elements, or an element in relation to another. In this chapter, we'll look at ways to select content based on other criteria, such as attributes and their values, how the user is interacting with the element, where it appears in the document tree, and more. We'll also finally look at using external style sheets, which is by far the most common and recommended method for attaching styles to a document—I've saved the best for last!

SELECTORS YOU'VE LEARNED SO FAR

Before we dive in, let's take a look at the selectors you should be comfortable using at this point. The capital letter "E" indicates an element name in the selector structure examples:

Element type selector

Matches the name of an element.

```
E {property: value;}
```

Grouped (compound) selectors

Matches all the elements in a comma-separated list.

```
E1, E2, E3 {property: value;}
```

Descendant selector combinator

Matches element E2 only if it is a descendant of element E1.

```
E1 E2 {property: value;}
```

Child selector combinator

Matches any element E2 that is a direct child of element E1.

```
E1 > E2 {property: value;}
```

Next-sibling selector combinator

Matches any element E2 that *immediately follows* element E1, where E1 and E2 share the same parent element.

```
E1 + E2 {property: value;}
```

Subsequent-sibling selector combinator

Matches any element E2 that is preceded by element E1, where E1 and E2 share the same parent element.

```
E1 ~ E2 {property: value;}
```

ID selector

Matches the value of the **`id`** attribute in an element. Because **`id`** values are unique in a document, the element name may be omitted.

```
E#id {property: value;}
#id {property: value;}
```

Class selector

Matches the value of the **`class`** attribute in all elements or a specific element. When used without an element name, it selects all elements with the provided class value.

```
E.class {property: value;}
.class {property: value;}
```

Universal selector

Matches all elements.

```
* {property: value;}
```

Now let's add to your collection!

ATTRIBUTE SELECTORS

Another way to target content based on its markup is to use an **attribute selector**. As its name indicates, an attribute selector targets elements based on attribute names or the values of those attributes. This provides a lot of flexibility for selecting elements without needing to add a lot of extra **`class`** or **`id`** markup.

By Attribute Name

The **simple attribute selector** (**`[`***`att`***`]`**) targets elements that contain a particular attribute, regardless of its value. The following example selects any image that has a **`title`** attribute and applies a border to it:

```
img[title] {border: 3px solid;}
```

You could also specify just the attribute name to capture all elements in the document that have a **title** attribute:

```
[title] {border: 3px solid;}
```

You can chain attribute selectors together as well, to narrow down the target. This example matches only images that have both **title** and **loading** attributes:

```
img[title][loading] {border: 3px solid;}
```

Because attribute names are case-insensitive in HTML, **[title]** would also match elements with the attribute **TITLE** (see **Note**).

By Attribute Value

The remaining attribute selectors target elements based on the *values* of their attributes. The simplest of these—the **exact attribute value selector**, (**[***att***="***value***"]**)—selects elements with a specific value for the attribute. This selector matches images with *exactly* the **title** value "first grade":

```
img[title="first grade"] {border: 3px solid;}
```

You can target just one word in an attribute value as well. For example, if you wanted to select images with the titles "first grade", "second grade", and "third grade", you could use a **partial attribute value selector** to look for the word "grade," and all instances would be selected. The partial value selector is indicated with a tilde (**~**), as in **[***att***~="***value***"]**), and looks for a specified value in a list of values that are separated by character spaces. Here's how that example would be written in the style sheet:

```
img[title~="grade"] {border: 3px solid;}
```

The remaining value selectors allow you to select a string of characters (a **substring**) within a value. This is slightly different from the partial attribute value selector, which finds parts of values when they are separated by spaces (as in "first grade" or "second grade"). They are:

Beginning substring attribute value selector (^)

element[*attribute*^="*first part of the value*"]

Matches elements whose specified attribute values *start* with the string of characters in the selector. Beginning substrings are indicated by a caret (**^**) character after the attribute name. This example applies the style only to images that are found in the */images/icons* directory:

```
img[src^="/images/icons"] {border: 3px solid;}
```

Ending substring attribute value selector ($)

element[*attribute*$="*last part of the value*"]

Matches elements whose specified attribute values *end* in the string of characters in the selector. Ending substrings are indicated by the dollar

NOTE

*Because other markup languages that use CSS may be stricter and define attribute names as case-sensitive, CSS provides modifiers that make attribute names case-insensitive (***i***) or case-sensitive (***s***) regardless of the default. The modifier is placed within the attribute selector brackets, like so:*

[*att* **i**]

When used with HTML, the modifiers are more useful for specifying the case sensitivity of value names.

(**$**) symbol. This example applies a style to only the **a** elements that link to PDF files:

```
a[href$=".pdf"] {border-bottom: 3px solid;}
```

Arbitrary substring attribute value selector (*)

element[*attribute**="*any part of the value*"]

Looks for the provided text string in any part of the attribute value specified (indicated by an asterisk, *). This rule selects any image that contains the word "November" somewhere in its **alt** text:

```
img[alt*="November"] {border: 3px solid;}
```

Hyphen-separated attribute selector (|)

element[*attribute*|="*value*"]

Targets elements with an attribute whose value matches "*value*" exactly or begins with "*value*" followed immediately by a hyphen. This selector is useful for targeting hyphen-separated language codes. For example, this example matches any link that points to a document written in a variation of the English language (**en**), with an attribute value of **en-us** (American English), **en-in** (Indian English), **en-au-tas** (Australian English), and so on:

```
[hreflang|="en"] {border: 3px solid;}
```

There's no better way to learn than by doing, so why not take some time to try out a few attribute selector types in EXERCISE 14-1?

PSEUDO-CLASS SELECTORS

Have you ever noticed that a link is often one color when you click or tap it, and then another color when you go back to that page? That's because, behind the scenes, your browser is keeping track of which links have been clicked (or "visited," to use the lingo). The browser keeps track of other states too, such as whether the user's cursor is over an element (hover state), whether an element is the first of its type in its parent element, whether it's the first or last child element, and whether a form element has been checked or disabled, just to name a few.

In CSS, you can apply styles to elements in these states by using a special kind of selector called a **pseudo-class selector**. It's an odd name, but you can think of it as though elements in a certain state belong to the same class. In this case, however, the class name isn't in the markup—it's something the browser just keeps track of. So, it's *kinda* like a class...it's a *pseudo*-class.

Pseudo-class selectors are indicated by the colon (**:**) character. They typically go immediately after an element name—for example, **li:first-child**—but they may also be used alone to select multiple element types.

EXERCISE 14-1. Attribute selectors

In this exercise, you'll get some experience with basic attribute selectors, which you'll use to add background colors to form inputs. Open the document *specials.html* (included with the materials for this chapter, available at *learningwebdesign.com/6e/materials*) in a text or code editor. There is an embedded style sheet with rules for formatting the text so you can concentrate solely on selectors.

I've organized the style rules into sections appropriate to the exercises in this chapter and labeled them using CSS comments (for example, `/* FORM STYLES */`). This is a common method for keeping style sheets organized and tidy, and it helps when several people are collaborating on the same CSS document. Do your best to keep the new style rules you add in these exercises in their relevant sections, as noted. Be aware, though, that this is just a recommendation. In general, there is no "right" way to organize a style sheet; you may have your own preferences, or you may need to follow the conventions of the team you are working with.

Without further ado, it's time to select the VIP sign-up form inputs for styling.

1. The first thing we'll do is set the background of the Birth Month `<select>` menu to the color "peachpuff" by targeting its `size` attribute with a simple attribute selector. Add the following style rule to the "FORM STYLES" section of the style sheet:

   ```
   [size] {
     background-color: peachpuff;
   }
   ```

   Note: I am using names from the CSS Color Module in the exercises because they are intuitive and easier to copy than a string of numbers, but know that they aren't used much in real web development. As discussed in **Chapter 13, Colors and Backgrounds**, numeric values (RGB or HSL) are the norm.

   Because there is no element name specified, this rule would select all the elements in the document that have a `size` attribute (in this document, there is only one). You may be thinking, "Couldn't I just use the `<select>` element as a selector?" Yes, that would do the same thing, but you will find that there are often several ways to select an element, and this is an opportunity to see how an attribute selector works.

2. Next, make the background of the Email input the "lightsalmon" color. It uses the `name` attribute with its value set to "email", so we can use that name/value pair in an exact value selector (=) to target the element:

   ```
   [name="email"] {
     background-color: lightsalmon;
   }
   ```

3. Now let's add a "salmon" background color to both the First Name and Last Name text inputs. Both of their `name` attributes have values that end with "-name", so we'll use an ending substring value selector (indicated by `$`) to apply styles to both of them:

   ```
   [name$="-name"] {
     background-color: salmon;
   }
   ```

Save the file and take a look at it in a browser. The resulting form fields should look like the sample in FIGURE 14-1. Hang on to the *specials.html* file, because we'll be using it for the remaining exercises in this chapter.

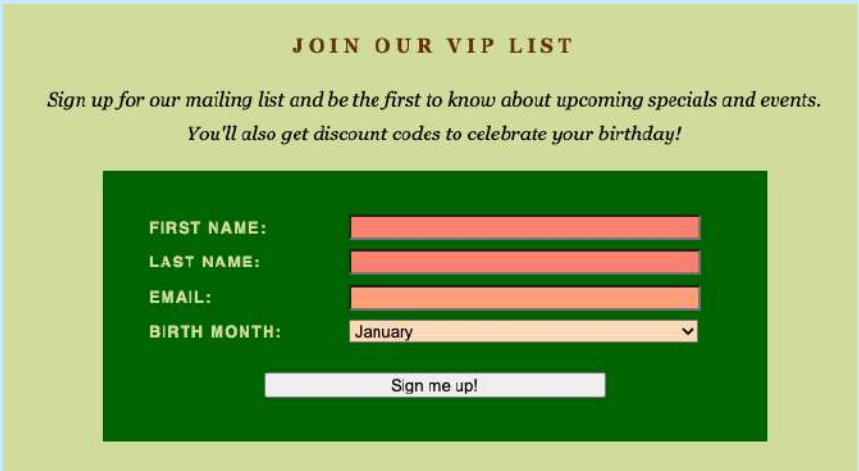

JOIN OUR VIP LIST
Sign up for our mailing list and be the first to know about upcoming specials and events.
You'll also get discount codes to celebrate your birthday!
FIRST NAME:
LAST NAME:
EMAIL:
BIRTH MONTH:
January
Sign me up!

FIGURE 14-1. Tasty salmon and peachpuff form input colors added with attribute selectors

There are dozens of pseudo-classes in CSS Selectors Module Level 4, many of which are still considered experimental as of this writing. In this section, I'll introduce you to the most useful and best-supported pseudo-class selectors as a solid starter kit. You can explore the cutting-edge selectors as you gain more experience. The full list of Level 4 CSS selectors, with descriptions, can be found in **Appendix C, CSS Selectors, Level 4**.

Location Pseudo-Classes

A good place to start learning about how pseudo-classes work is with the set of **location pseudo-class selectors**, which are related to links. Here's the complete list of these selectors from the Level 4 module—this should give you a good overview (see **Note**):

`:link`	Selects unvisited links
`:visited`	Selects links that have already been visited
`:target`	Selects the element that is the target of a same-page fragment link (indicated by `#`)
`:scope`	Matches elements in the "scoping root" of the document, which is the same as `:root` unless otherwise established with JavaScript
`:any-link`	Selects all links, regardless of whether or not they've been visited
`:local-link`	Selects links that link to the current document, such as to a fragment (indicated by `#`)
`:target-within`	Selects an element that is or contains a target of a same-page link

BROWSER SUPPORT NOTE

Browser support for `:any-link`, `:local-link`, *and* `:target-within` *ranges from "spotty" to "totally unsupported" at the time of writing. Support changes over time, however, so I recommend that you check CanIUse.com to verify the current reliability.*

■ USABILITY TIP

When you alter the appearance of links and visited links, be sure that they still look like links. Furthermore, it is recommended that you don't rely on color alone to indicate that text is clickable.

The two location pseudo-class selectors you're most likely to use when you're getting started with CSS are `:link` and `:visited`. By default, browsers typically display linked text as blue and links that have been visited as purple. You can change that with a few style rules, but be aware that there are limitations on what properties may be applied to `:visited` links, as explained in the **"Visited Links and Security"** sidebar.

In these examples, I've changed the color of unvisited links to maroon and visited links to gray. It is common for visited links to be a more muted color than unvisited links:

```
a:link {
  color: maroon;
}
a:visited {
  color: gray;
}
```

User Action Pseudo-Classes

The set of **user action pseudo-classes** pertain to states that result from direct user actions, such as pointing the mouse at an element or tabbing through form fields. The full list is presented below, but we will concentrate on the selectors you are most likely to use right away: `:focus`, `:hover`, and `:active`.

`:focus`	Applies when the element is selected and ready for input
`:hover`	Applies when the mouse pointer is over the element
`:active`	Applies when the element (such as a link or button) is in the process of being clicked or tapped
`:focus-visible`	Applies to elements that have focus, but only when it matches the browser's determination of whether the element requires focus (this prevents styling elements that don't normally get highlighted when they are focused, such as links)
`:focus-within`	Applies to an element if it or any of its descendants has focus

Focus state

If you've ever used a web form, you should be familiar with how a browser visually emphasizes a form element when you select it. When an element is highlighted and ready for input, it is said to have "focus." The **`:focus`** selector lets you apply custom styles to elements when they are in the focused state.

In this example, when a user selects a text input, it gets a yellow background color to make it stand out from the other form inputs:

```
input:focus {background-color: yellow;}
```

Hover state

The **`:hover`** selector targets elements while the user's mouse pointer is directly over them. You can use the hover state with any element, although it is most commonly used with links to give the user visual feedback that an action is possible. Hover states are also used to trigger pop-up menus for navigation or for revealing more information about an object on the page.

This rule gives links a light pink background color while the mouse hovers over them:

```
a:hover {
  color: maroon;
  background-color: #ffd9d9; /* light pink */
}
```

In **Chapter 12, Formatting Text**, you saw how to use the **`text-decoration`** property to turn off underlines under links. You could then use the **`:hover`** selector to make the underlines appear only "on hover":

```
a:hover {
  text-decoration: underline;
}
```

It is important to note that there is no true hover state on touchscreen devices such as smartphones and tablets. See the sidebar **"Hover on Touch Devices"** for more information.

Visited Links and Security

Browsers keep track of what links have been visited, but for some users, a record of their visited links (which could be stolen by a malicious site) may be undesirable. For people in regions with severe restrictions on viewing online content, that record in the wrong hands could even be life-threatening. When it was discovered that visual styles applied to visited links, as well as the methods browsers use to keep track of them, could be used to track users' viewing histories, browsers made changes to how they handle visited links.

The first change was to limit the visual presentation properties that can be applied to visited links. Style rules with **`:visited`** pseudo-class selectors may use only the following properties: **`color`**, **`background-color`**, **`border-color`** (and individual side border properties), and **`outline-color`**. Any other property will be ignored. Furthermore, you cannot use any value that makes the link transparent, including the **`transparent`** keyword and RGB and HSL color values with alpha transparency.

Under the hood, the DOM mechanism that keeps track of what links have been visited will always return a "not visited" state, even when visited styles are displayed on the screen. This keeps browsing history hidden at the DOM level as well.

Active state

The `:active` selector applies styles to an element while it is in the process of being activated. In the case of a link, it is the style that is applied while it is being clicked or while a fingertip is in contact with it on a touchscreen. This style may be displayed only for an instant, but it can give a subtle indication that something has happened. The rule in this example brightens up the color for the active state (from maroon to red):

```
a:active {
  color: red;
  background-color: #ffd9d9;
}
```

Better Link Feedback with Pseudo-Classes

Web designers commonly provide styles for linked, visited, focused, hover, and active states because it is an easy way to give a nice bit of feedback at every stage of clicking or tapping a link, and it usually improves on the

Hover on Touch Devices

On a desktop computer, the mouse pointer can hover over elements on the screen, but touch devices respond only when the screen is actually touched. This can make hover effects problematic on smartphones and tablets.

When hover effects are applied to a link (an `a` element), mobile operating systems may display the hover state styles after a long tap. To follow the link, the user must tap again. Other hover-triggered elements, such as pop-up menus, may get stuck open, requiring the user to tap elsewhere or reload the page to clear it (not a good user experience, and a deal-breaker for some designs).

Always including `:focus` and `:active` state styles along with the `:hover` styles may help in some situations. A better solution is to create a separate set of styles for devices that allow hovering and another set of styles for those that don't, and use media queries to detect whether the device supports the hover state (`@media (hover: hover)` or `@media (hover: none)`). The query then serves the appropriate styles based on how the device responds to the test. Media queries are introduced in **Chapter 19, Responsive Web Design**. Once you're up to speed on media queries, Kevin Powell explains how to deal with hover in his video "Dealing with Hover on Mobile" (*youtube.com/watch?v=uuluAyw9AI0*).

You could also use JavaScript to create the desired effect for mobile devices. JavaScript solutions are beyond the scope of this chapter, but these resources can get you started (some knowledge of JavaScript is required):

- "A Guide to Hover and Pointer Media Queries" by Cristian Diaz (*smashingmagazine.com/2022/03/guide-hover-pointer-media-queries/*).
- "Custom Mouse Hover Effect with JavaScript" on Coding Artist (*codingartistweb.com/2023/12/custom-mouse-hover-effect-with-javascript/*).
- Stack Overflow (*StackOverflow.com*). This is a forum where programmers can ask questions and get help from fellow programmers. Try searching for "hover states on touch devices"—you'll find a lot of solutions, but also likely some dead ends.

browser defaults. In fact, users have come to expect this feedback: seeing at a glance which links have been followed, having links do something when they point at them, and receiving confirmation when the links are successfully clicked.

When you apply styles to **a** elements with location and user-action pseudo-classes, the order in which they appear is important for them to function properly. For example, if you put **:link** or **:visited** last, it will override the other states, preventing them from appearing. The required order for link pseudo-classes is **:link**, **:visited**, **:focus**, **:hover**, **:active** (LVFHA, which you can remember with LoVe For Hairy Animals, or the mnemonic device of your choice).

The required order for pseudo-classes is:
:link
:visited
:focus
:hover
:active

It is recommended that you provide a **:focus** style for users who use the keyboard to tab through links on a page rather than clicking with a mouse. Applying the same style used for **:hover** is common, although not required. However, if you do not want ordinary links to get styled, you can use the **:focus-visible** selector instead, which applies styles only to elements browsers highlight by default, like form inputs.

To sum things up, the link-related pseudo-classes we've looked at so far in this section would look like this when collected in a style sheet—FIGURE 14-2 shows the results:

```
a { text-decoration: none; }  /* turns underlines off for all links */
a:link { color: maroon; }
a:visited { color: gray; }
a:focus { color: maroon; background-color: #ffd9d9; }
a:hover { color: maroon; background-color: #ffd9d9; }
a:active { color: red; background-color: #ffd9d9; }
```

In EXERCISE 14-2, you'll get a chance to try out link selectors yourself.

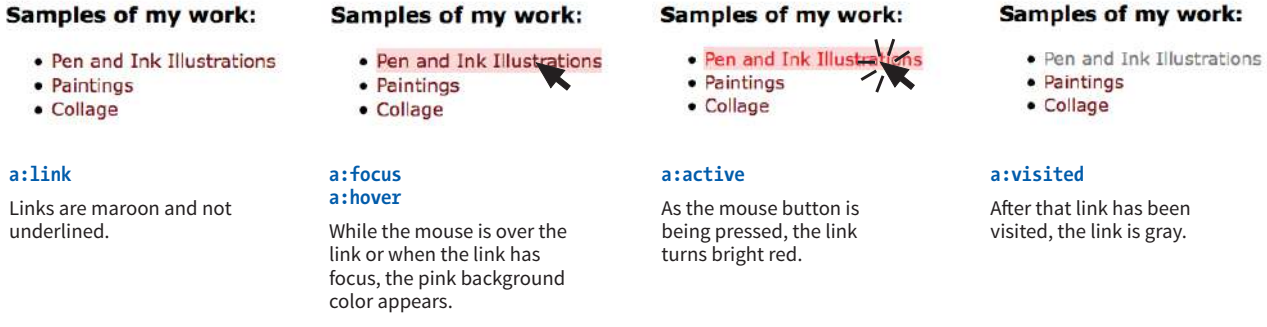


FIGURE 14-2. Changing the colors and backgrounds of links with pseudo-class selectors

EXERCISE 14-2. Pseudo-class selectors for links

In this exercise, we'll override the browser's default colors for the list of links in the header of the *specials.html* page. The underlines have been turned off for links (**`text-decoration: none`**), and I've added a rounded border and a white background at 25% opacity. We'll add colors to each link state to provide feedback in the link-clicking process. Add the rules in this exercise to the "LINK STYLES" section of the style sheet.

1. Write a rule that makes unvisited links the same purple as the **`h1`** (**`#993399`**).
2. Make visited links a muted purple (**`#937393`**). Remember, it's common for visited links to be a less vibrant color than unvisited links.
3. For the hover state (when the mouse is placed over links), make the text a brighter purple (**`#C700F2`**) and add a white background color (**`#fff`**). It will look a little like the links are lighting up when the mouse is pointing at them. Apply the same declarations to links that are in focus.
4. For the state where the link is actively being clicked or tapped, keep the white background color but make the text brighten up even more to "fuschia". Make sure that all of your pseudo-classes are in the right (LVFHA) order. You can check your work against this finished code:

```
a:link {
  color: #993399;
}
a:visited {
  color: #937393;
}
a:focus, a:hover {
  color: #C700F2;
  background-color: #fff;
}
a:active {
  color: fuchsia;
  background-color: #fff;
}
```

When you're finished, save the document and take your new and improved links for a spin (FIGURE 14-3).

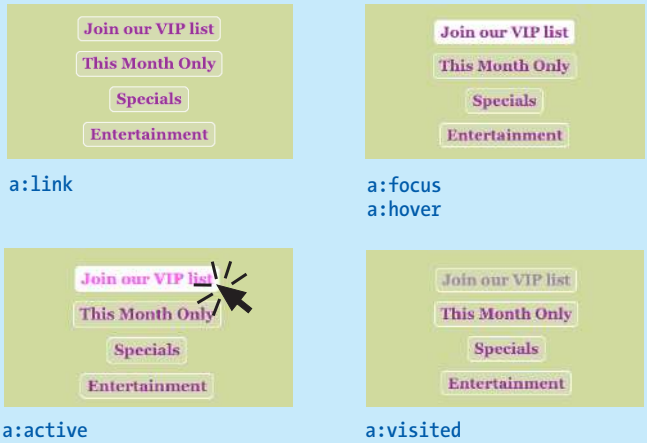


FIGURE 14-3. Colors applied to unvisited and visited links and the hover, focus, and active states give the user important feedback.

Tree-Structural Pseudo-Classes

Another way to target elements in the document without specifically calling them out by name is to use a **tree-structural pseudo-class**. These pseudo-classes select an element or elements based on where they appear in the document tree (or the DOM, to use the technical term). Here's an overview, but we'll look at them in more detail in this section and you'll get a chance to try them out along the way:

`:root`	Applies to the root element of a document (**`html`** in HTML)
`:empty`	Selects elements with no content
`:nth-child()`	Selects an element that is the *n*th child of its parent element (indicated by an integer, a numeric expression, or the keywords **`odd`** or **`even`**)
`:nth-last-child()`	Selects an element that is the *n*th child of its parent element, starting from the last one (indicated by an integer, a numeric expression, or the keywords **`odd`** or **`even`**)
`:first-child`	Selects an element that is the first child of its parent element
`:last-child`	Selects an element that is the last child of its parent element
`:only-child`	Selects an element that is the only child of its parent element
`:nth-of-type()`	Selects the *n*th element of its type (indicated by an integer or numeric expression)
`:nth-last-of-type()`	Selects the *n*th element of its type (indicated by an integer or numeric expression), starting from the last one
`:first-of-type`	Selects an element that is the first sibling of its type
`:last-of-type`	Selects an element that is the last sibling of its type
`:only-of-type`	Selects an element that is the only sibling of its type

The **`:root`** selector applies styles to the root element of the document, which is **`html`** in HTML documents, but keep in mind that CSS may be used for documents in other markup languages, so that's not always a given. As its name suggests, **`:empty`** applies to any element that does not have any content, which could be the case when an element acts as a placeholder for content that gets added dynamically, as well as other scenarios.

The more interesting structural pseudo-class selectors are the **child-indexed** and **typed child-indexed** selectors. To get acquainted with them, we'll try a few out to select list items in this ordered list of fruits:

```
<ol>
  <li class="red yellow">Apples</li>
  <li class="red green">Grapes</li>
  <li class="yellow">Bananas</li>
  <li class="yellow">Mangoes</li>
  <li class="blue">Blueberries</li>
  <li class="red">Lychees</li>
```

```
    <li class="orange">Oranges</li>
    <li class="yellow">Lemons</li>
    <li class="red">Cherries</li>
    <li class="orange">Nectarines</li>
</ol>
```

Simple child-indexed selectors

I bet you can guess which list item would get a pink background with this selector:

```
li:first-child {background-color: pink;}
```

Did you say "1. Apples"? That's right. That's the **li** that is the first child of its parent element, **ol**. How about a slightly fancier example? To select the first "orange" fruit in the list ("7. Oranges"), we could use **:first-child** with a class selector:

```
.orange:first-child {background-color: pink;}
```

Based on that, it should be clear that **:last-child** would select "10. Nectarines," as it is the last child element of the **ol**. To select "9. Cherries," we could use **.red:last-child**, because it is the last child element of **ol** that has the class "red".

The **:only-child** selector selects an element when it is the *only* child element of its parent, which doesn't apply to our list code example (nothing would be selected).

nth-child selectors

The **:nth-child()** and **:nth-last-child()** pseudo-class selectors offer more fine-tuned control over selecting child elements. The parentheses at the end indicate that these selectors are functions to which you pass an argument. The selector then uses the argument value to calculate which child element to select. The **:nth-** selectors can take an integer (whole number), the keywords **odd** or **even**, or a numeric expression (I'll explain what that is in a moment) as arguments.

Let's start with the simplest method. Using the previous ordered list example, to select the third list item, "3. Bananas," just pass the function the integer 3:

```
li:nth-child(3) {background-color: pink;}
```

CSS counts the elements starting with 1, which sounds obvious, but it's worth pointing out if you are familiar with JavaScript, which starts counting at 0.

You can also select all of the odd or even children of a parent element. This might come in handy for applying alternating styles or colors to table rows or other lists to help with readability. FIGURE 14-4 shows what happens when we select all of the odd-numbered list items in our list of fruits for styling:

```
li:nth-child(odd) {background-color: pink;}
```

The third way you can use the `:nth-child()` pseudo-class selector is to pass it a numeric expression, which is like a tiny formula. The "n" in the expression indicates a calculation. In this example, the expression `3n` indicates that we want to select every third child element, starting with the first child. In our list, this selector grabs "3. Bananas," "6. Lychees," and "9. Cherries" (FIGURE 14-5 A):

```
li:nth-child(3n) {background-color: pink;}
```

To start at a position other than the first child, add the position after a + sign. This example starts counting with the second child, then selects every third child after that. As a result, it selects items 2, 5, and 8 (FIGURE 14-5 B):

```
li:nth-child(3n+2) {background-color: pink;}
```

Providing a negative numeric expression changes the direction of the selection. For example, if we wanted to select the fourth item and every item before it, we would use this selector (FIGURE 14-5 C):

```
li:nth-child(-n+4) {background-color: pink;}
```

The `nth-last-child` selector works similarly, but it starts counting at the end of the list and works backward toward the beginning. The selector in this example starts counting at the third-from-last element ("8. Lemons") and selects every second item above it in the list (FIGURE 14-5 D):

```
li:nth-last-child(2n+3) {background-color: pink;}
```

Numeric expressions like these can be a little confusing at first. Just remember to start with the number after the plus sign, which says where to start counting, then use the number before the `n` to determine how many child elements to jump over each time. If there is no plus sign, start at the first element.

I've put together EXERCISE 14-3 for you to get some practice using `-child` pseudo-selectors.

1. Apples
2. Grapes
3. Bananas
4. Mangoes
5. Blueberries
6. Lychees
7. Oranges
8. Lemons
9. Cherries
0. Nectarines

FIGURE 14-4. Selecting only the odd-numbered children in our ordered list for styling

FIGURE 14-5. Using functional notations with the `:nth-child()` pseudo-selector

EXERCISE 14-3. Child-indexed pseudo-class selectors

There is a Daily Specials table on the Black Goose Bistro Specials page (*specials.html*) that will allow us to have some fun with the **-child** selectors. We'll be adding background colors to table rows (**tr** elements) to make the table easier to parse for users with visual browsers. You can save your file and look at it in the browser after each step to check your progress. The initial and finished versions of the table are shown in FIGURE 14-6.

1. Let's make the row of header cells a darker color to make it stand out. The "tomato" color should do nicely. Because the header row is the first child within the **table** element, we can use the **:first-child** selector:

   ```
   tr:first-child {background-color: tomato;}
   ```

2. Now let's give the rows alternating colors. Start by giving all of the even-numbered rows a "wheat"-colored background with the **:nth-child** selector and the keyword **even**:

   ```
   tr:nth-child(even) {background-color: wheat;}
   ```

3. Looking good! Next, fill the backgrounds of the odd rows with a subtly contrasting "blanchedalmond" color:

   ```
   tr:nth-child(odd) {background-color: blanchedalmond;}
   ```

4. Save the document and look at the page in your browser. Oops! That rule overrode the color we had set for the first row, so using the **odd** keyword isn't exactly what we want. Hide that rule by putting it in a CSS comment (add **/*** before the rule and ***/** after it), so it will be hidden but you won't have to retype it if you change your mind.

   Let's try it again using a numeric expression that allows more fine-tuned control. We want the "blanchedalmond" color to apply to the third **tr** element, and then to every second row thereafter. Here's the rule to make that happen:

   ```
   tr:nth-child(2n+3) {background-color: blanchedalmond;}
   ```

5. Because the restaurant is closed on Sundays, we'll give that row a dimmed-down background color ("darksalmon"). We can use the **:nth-last-child()** selector to start at the end of the list of **tr** elements and the number 1 to choose only the first **tr** from the end:

   ```
   tr:nth-last-child(1) {background-color: darksalmon;}
   ```

   Of course, in this instance, we could have used the **:last-child** selector for the same effect, but now you have experience starting from the last element. Feel free to try different arguments for the **:last-child** selector to get a feel for how it works.

That's it for the table. Does yours look like the one in the "After" shot in FIGURE 14-6? And did you happen to notice that all of the colors used for this restaurant page have food-related names? That's the kind of attention to detail I doubt you'll find in other web design books!

Before

After

FIGURE 14-6. The Black Goose Bistro Daily Specials table before and after applying background colors with tree-structural pseudo-classes

Typed child-indexed selectors

You should now have a basic understanding of how the **`-child`** pseudo-selectors work. In our ordered list example, all of the sibling elements were list items (**`li`**), but be aware that the **`-child`** selectors count through *all* the sibling elements that share a parent element, regardless of type. Imagine an article that has a mix of headings, paragraphs, and blockquotes in the flow of content—an **`:nth-child(even)`** selector would target every other element, whatever it happens to be.

If you'd like to be more specific about which type of element you want to select, use one of the **typed child pseudo-selectors** (**`:first-of-type`**, **`:last-of-type`**, **`:only-of-type`**, **`:nth-of-type()`**, and **`:nth-last-of-type()`**). They work the same as the **`-child`** selectors we looked at previously in terms of syntax, but they limit the target to just the element type you specify.

I'll use a definition list (**`dl`**) to demonstrate how typed child selectors behave. Just like the **`li`** elements in the ordered list, all of the **`dt`** and **`dd`** elements are siblings within their parent **`dl`** element:

```
<dl>
  <dt>Red fruit</dt>
  <dd>Apples</dd>
  <dd>Grapes</dd>
  <dd>Lychees</dd>
  <dd>Cherries</dd>

  <dt>Yellow fruit</dt>
  <dd>Apples</dd>
  <dd>Bananas</dd>
  <dd>Mangoes</dd>
  <dd>Lemons</dd>

  <dt>Orange fruit</dt>
  <dd>Oranges</dd>
  <dd>Nectarines</dd>
  <dd>Mangoes</dd>

  <dt>Blue fruit</dt>
  <dd>Blueberries</dd>
</dl>
```

To apply styles to all of the even **`dd`** elements within the **`dl`** and avoid styling the **`dt`** elements, use the **`:nth-of-type()`** selector as follows (FIGURE 14-7):

```
dd:nth-of-type(even) {background-color: pink;}
```

We've really only scratched the surface of using tree-structural pseudo-class selectors. For a more detailed explanation, along with examples and browser support information, see the MDN Web Docs "CSS Selectors" reference guide (*developer.mozilla.org/en-US/docs/Web/CSS/CSS_selectors*). You will also find the article "How to Use CSS3 Pseudo-Classes" by Richard Shepherd to be helpful (*oreil.ly/azcov*).

Before we move on, why not try your hand at using the **`-of-type`** pseudo-selectors in EXERCISE 14-4?

FIGURE 14-7. Using the `:nth-of-type()` selector to limit selections to `dd` elements

EXERCISE 14-4. Typed structural pseudo-classes

In this exercise, we'll add background colors to the **h3** headings in the "This Month Only" section of the page to visually emphasize the subsections. If you look at the markup, you'll see that this section is marked up as a **section** element with the **id** "monthly" that acts as the container for a number of heading and paragraph elements. It's important to note that all of the elements it contains are siblings—that is, direct children of the **#monthly** section.

1. We're going to give all of the odd **h3** elements the background color "chocolate" but first, let's do a little experiment. In the "MONTHLY STYLES" section of the style sheet, add a rule that applies the style to **h3**s using the **:nth-child** selector:

   ```
   #monthly h3:nth-child(odd) {
     background-color: chocolate;
   }
   ```

   Save the file and open it in a browser. You should see that only the first and last **h3**s have been colored. That's because the selector is selecting every odd element among *all* of the children in the **#monthly** section. Those other **h3**s happen to be in even positions. So, let's change the selector to **:nth-of-type()** to get the effect we're after:

   ```
   #monthly h3:nth-of-type(odd) {
     background-color: chocolate;
   }
   ```

   Save the file and refresh the page in your browser. You should see that the first, third, and fifth **h3**s are properly colored, because now the browser is looking for odd-numbered elements only among the **h3** children in the section.

2. The even-numbered **h3**s are looking a little weak now, so we'll give them a background color too. The resulting styles are shown in FIGURE 14-8:

   ```
   #monthly h3:nth-of-type(even) {
     background-color: olive;
   }
   ```

   Note: A more streamlined approach with the same alternating effect would be to set a color on all **h3**s, then override it with a color set on just the odd or even ones. Again, there is usually more than one way to achieve an effect with selectors. In this exercise, I wanted you to have the chance to use both the **odd** and **even** keywords.

3. This exercise is basically finished, but for fun, you could make the **h3**s the colors of the rainbow by targeting them one at a time with an integer (whole number) as the argument. You can decide if you want to add these rules to the style sheet—honestly, it's not the best look, but it does give you more practice:

   ```
   #monthly h3:nth-of-type(1) {background: violet;}
   #monthly h3:nth-of-type(2) {background: deepskyblue;}
   #monthly h3:nth-of-type(3) {background: greenyellow;}
   #monthly h3:nth-of-type(4) {background: orange;}
   #monthly h3:nth-of-type(5) {background: red;}
   ```

What is significant in all of these examples is that we were able to zero in on specific elements without adding any extra markup, such as classes or extra **div**s. That's a superpower of pseudo-classes.

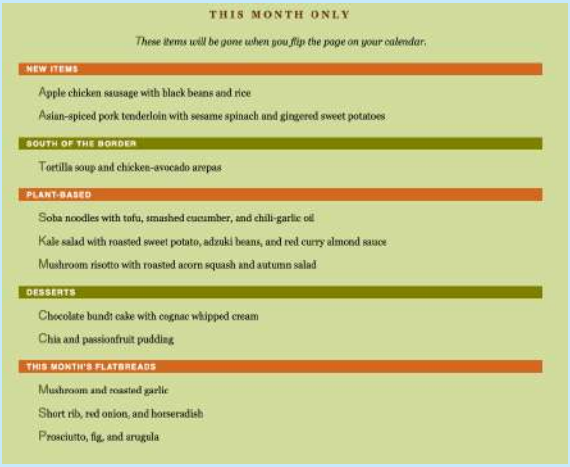

THIS MONTH ONLY
These items will be gone when you flip the page on your calendar.
NEW ITEMS
Apple chicken sausage with black beans and rice
Asian-spiced pork tenderloin with sesame spinach and gingered sweet potatoes
SOUTH OF THE BORDER
Tortilla soup and chicken-avocado arepas
PLANT-BASED
Soba noodles with tofu, smashed cucumber, and chili-garlic oil
Kale salad with roasted sweet potato, adzuki beans, and red curry almond sauce
Mushroom risotto with roasted acorn squash and autumn salad
DESSERTS
Chocolate bundt cake with cognac whipped cream
Chia and passionfruit pudding
THIS MONTH'S FLATBREADS
Mushroom and roasted garlic
Short rib, red onion, and horseradish
Prosciutto, fig, and arugula

FIGURE 14-8. Using the **:nth-of-type()** selector to target only the **h3**s in a section

Logical Pseudo-Class Selectors

There's one more pseudo-class that you might find helpful right away: the **`:not()`** selector, also called the **negation selector**. You provide a selector as the argument (between parentheses), and it selects elements that do *not* match that selector. This is one of a small collection of **logical pseudo-class selectors** in CSS Selectors Module Level 4.

`:not()`	Selects elements that do not match a list of selectors
`:is()`	Selects any element that can be selected by one of the selectors in the list
`:where()`	Selects any element that can be selected by one of the selectors in the list, but sets the specificity of all selectors to 0
`:has()`	Selects an element using relative selectors, which is a way to select the parent or sibling(s) of the reference element

A few examples should make the use of the **`:not()`** selector clear. Take a look back at the ordered list of fruit that we used to explore the **`-child`** selectors in **"Tree-Structural Pseudo-Classes."** Each list item has been marked up with a **`class`** attribute that identifies the color of the fruit. To select all the items *except* the ones classified as "yellow", provide the **`.yellow`** class selector (remember that class selectors are indicated by a period) as the argument of the **`:not()`** selector:

```
li:not(.yellow) {background: papayawhip;}
```

Here's another example that selects all the items in the list except the first one:

```
li:not(:first-child) {background: papayawhip;}
```

Finally, let's try one that puts together a number of the selector types you've learned about so far. Say you wanted to make all the elements within a **`div`** with the **`id`** "container" red, *except* the paragraphs and blockquotes that are direct children of the **`div`**. The **`>`** between **`#container`** and **`:not()`** makes this a child combinator, so this rule looks inside the container for the elements to match (or not match, in this case). The argument of **`:not()`** is a grouped selector:

```
#container > :not(p, blockquote) {color: red;}
```

This is still a relatively straightforward example, but you can begin to see how selectors can get quite complex.

The negation selector is a useful tool to have in your CSS selectors toolkit. It's a little quirky, so you might want to check out the tips on using it on the MDN Web Docs reference page first (***developer.mozilla.org/en-US/docs/Web/CSS/:not***). The MDN reference pages are a good place to learn about the other logical pseudo-classes as well, because we'll look at them only briefly here. The **`:is()`** selector takes a list of selectors as its argument and will match any element that can be selected by one of the selectors in the list. It is useful for reducing the amount of code for long lists of selectors. The **`:where()`** selector

works similarly, but has the difference of setting the specificity of the selectors to 0, which may be useful if one of the selectors in the list is adding unwanted specificity to the selector as a whole. Finally, the **:has()** selector makes selections relative to a reference element—for example, only **a** elements that contain an **img** child:

```
a:has(> img){margin: 0;}
```

EXERCISE 14-5 gives you a chance to get some hands-on experience with **:not()**.

EXERCISE 14-5. The :not() pseudo-class

This exercise uses the same Daily Specials table we worked on in EXERCISE 14-3. Look at the markup, and you will see that the first **td** in each row has been given the class "day". That's the hook I used to style the day names in those cells in a bold, sans-serif font. Let's give the cells in the table that are *not* classified as "day" a white background color with an opacity of 25% that lightens them but allows the background color of the row to show through (FIGURE 14-9):

```
td:not(.day) {
  background-color: rgb(255 255 255 / .25);
}
```

Remember that logical pseudo-selectors take a list of selectors as arguments, so I've provided the **.day** class selector to omit those cells.

I want to point out that there are other ways to target those cells. We could have used the **:last-child** selector, as those cells are the last children of the containing **tr** elements. Heck, we could have avoided adding the "day" class to the other cells and just used **:first-child** to select those. Again, the point is that there are often several ways to select the elements you are after—some better than others, and some just a matter of preference. You might also need to choose a selector that has more or less specificity than a conflicting selector (calculating specificity is covered in **Chapter 12**). This is why mastering CSS selectors is something of an art.

Now that you have **:not()** in your toolkit, here's another example of selector alternatives. In EXERCISE 14-3, we made all the odd rows in the table "blanchedalmond" except the first row, using the **tr:nth-child(2n+3)** selector. But we could also have used the simpler **odd** keyword and excluded first child **tr** element, like this:

```
tr:nth-child(odd):not(:first-child) {
  background-color: blanchedalmond;
}
```

That's a fun little example of how a chained selector can start to look a bit complicated, but you should be able to parse what it's doing if you've worked through the other exercises in this chapter. If it still feels overwhelming, don't worry; you'll build up your selector skills gradually with experience.

FIGURE 14-9. Applying a semi-transparent white background to cells that are *not* classified as "day"

Other Pseudo-Class Selectors

OK...28 pseudo-classes down, only 38 more to go! Well, I don't know about you, but that sounds like it would take a while, and I think we should probably be moving on. For the sake of brevity, I'll just list the remaining pseudo-classes here so you can get a general idea of the kinds of states the CSS Working Group finds it useful to be able to select. They are also listed with descriptions in **Appendix C**.

Keep in mind that CSS Selectors Module Level 4 is a work in progress, so it is possible that some of these will have been dropped and others added by the time you read this. It is also worth noting that many of these have little or no browser support as of this writing, so be sure to check ***CanIUse.com*** to see if they are viable options before using them.

Linguistic

- `:dir()`
- `:lang()`

Time-dimensional

- `:current`
- `:past`
- `:future`

Media playback state

- `:playing`
- `:paused`
- `:seeking`

Media loading state

- `:buffering`
- `:stalled`

Sound state

- `:muted`
- `:volume-locked`

Element display state

- `:open`
- `:closed`
- `:modal`
- `:fullscreen`
- `:picture-in-picture`

Input control

- `:enabled`
- `:disabled`
- `:read-only`
- `:read-write`
- `:placeholder-shown`
- `:autofill`
- `:default`

Input value

- `:checked`
- `:indeterminate`

Input value-checking

- `:blank`
- `:valid`
- `:invalid`
- `:in-range`
- `:out-of-range`
- `:required`
- `:optional`
- `:user-valid`
- `:user-invalid`

Grid-structural

- `:nth-col()`
- `:nth-last-col()`

PSEUDO-ELEMENT SELECTORS

Pseudo-classes aren't the only kind of pseudo-selectors. There are also **pseudo-element selectors** that select elements without markup (**`::first-line`** and **`::first-letter`**) and insert content into the document structure for styling (**`::before`** and **`::after`**). Pseudo-elements are indicated by a double colon (**`::`**) symbol to differentiate them from pseudo-classes (see **Note**).

NOTE

Although double colons are specified starting in CSS Level 3, browsers are required to support the legacy single-colon syntax going forward.

First Letter and Line

The first letter and first line of a text element are not identified with markup, so they are not proper elements, only "pseudo" elements. But you can select them on the fly with pseudo-element selectors:

`::first-line`

This selector applies a style rule to the first line of the specified element. The only properties you can apply, however, are as follows:

`color`
All `font` properties
All `background` properties
`word-spacing`
`letter-spacing`
`line-height`
`text-decoration`
`text-transform`
`text-shadow`
`text-decoration`
`text-decoration-color`
`text-decoration-line`
`text-decoration-style`
`vertical-align`

`::first-letter`

This applies a style rule to the first letter of the specified element. The properties you can apply are limited to the following (see **Note**):

`color`
All `font` properties
All `background` properties
All `padding` properties
All `margin` properties
All `border` properties
`letter-spacing`
`vertical-align` (if `float` is none)
`line-height`
`float`
`text-decoration`
`text-transform`
`text-shadow`
`text-decoration`
`text-decoration-color`
`text-decoration-line`
`text-decoration-style`
`box-shadow`
`vertical-align`

NOTE

There are a few properties in this list that you haven't seen yet. I'll cover the box-related properties (`margin`, `padding`, `border`) in **Chapter 15, Thinking Inside the Box.** *The `float` property is introduced in* **Chapter 16, Floating and Positioning.**

FIGURE 14-10 shows examples of the **`::first-line`** and **`::first-letter`** pseudo-element selectors:

```
p::first-line { letter-spacing: 9px; }

p::first-letter { font-size: 300%; color: orange; }
```

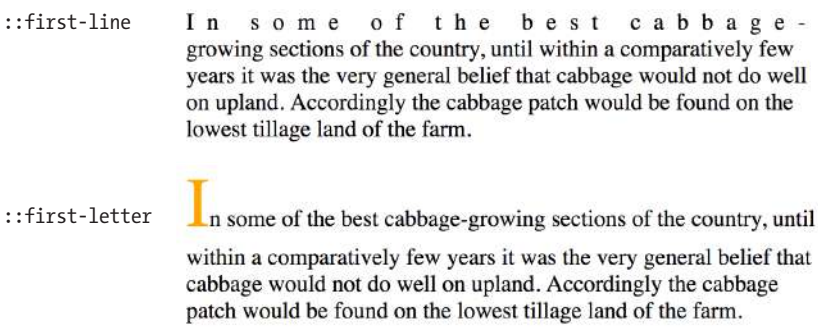

::first-line
In some of the best cabbage-growing sections of the country, until within a comparatively few years it was the very general belief that cabbage would not do well on upland. Accordingly the cabbage patch would be found on the lowest tillage land of the farm.
::first-letter
In some of the best cabbage-growing sections of the country, until within a comparatively few years it was the very general belief that cabbage would not do well on upland. Accordingly the cabbage patch would be found on the lowest tillage land of the farm.

FIGURE 14-10. Applying styles with the `::first-line` and `::first-letter` pseudo-element selectors

Generated Content with ::before and ::after

You've seen how browsers add bullets and numbers to lists automatically, even though they are not actually in the HTML source. That is an example of **generated content**, content that browsers insert on the fly. It is possible to tell browsers to generate content before or after any element by using the `::before` and `::after` pseudo-elements.

Generated content can be used to add icons before list items, to display URLs next to links when web documents get printed out, to add language-appropriate quotation marks around a quote, and much more. Here's a simple example that inserts an image by using the `url()` function before the paragraph and the text "Thank you." at the end of the paragraph. Compare the markup to what you see rendered in the browser (FIGURE 14-11).

THE MARKUP

```
<p class="warning">We are required to warn you that undercooked food is
a health risk.</p>
```

THE STYLES

```
p.warning::before {
  content: url(exclamation.png);
  margin-right: 6px;
}

p.warning::after {
  content: " Thank you.";
  color: red;
}
```

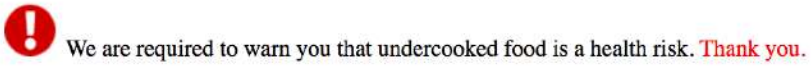


FIGURE 14-11. Generated content added with the `::before` and `::after` pseudo-selectors

There are a few things to note in this example:

- The pseudo-element selector goes immediately after the target element, without any space.
- The pseudo-element rule both inserts the content and specifies how it should be styled, in one declaration block.
- The `content` property, which provides the content you want inserted, is required. The selector won't do anything without it.
- If you want spaces between the generated content and the content from the source document, you must include the character spaces inside the value's quotation marks or apply a margin.
- If you want to insert an image, such as an icon or other mark, specify the URL without quotations marks, as shown in the first style rule.

When using generated content, keep in mind that whatever you insert does not become part of the document's DOM. It exists in the browser's display only and is not accessible to assistive devices like screen readers. It is therefore best to use generated content only for decorations and other "extras" that are not critical to your meaning and message. EXERCISE 14-6 gives you the opportunity to experience the magic and delight of pseudo-element selectors.

FURTHER READING

"Learning to Use the ::before and ::after Pseudo-Elements in CSS" by Louis Lazaris (*smashingmagazine.com/2011/07/learning-to-use-the-before-and-after-pseudo-elements-in-css/*) was written a while ago, but it still provides an excellent tutorial on generated content.

EXTERNAL STYLE SHEETS

Back in **Chapter 11, Introducing Cascading Style Sheets**, I told you that there are three ways to connect style sheets to an HTML document: inline with the `style` attribute, embedded with the `style` element, and as an external *.css* document linked to or imported into the document. In this section, we finally get to that third option.

External style sheets are by far the most common and powerful way to use CSS, because you can make style changes across an entire site simply by editing a single style sheet document. That's the advantage to having all the style information in one place, and not mixed in with the document source.

Furthermore, because a single style document is downloaded and cached by the browser for the whole site, there is less code to download with every document, resulting in better performance.

First, a little bit about the style sheet document itself. An external style sheet is a plain-text document with at least one style sheet rule. It may *not* include any HTML tags (there's no reason to include them, anyway). It may contain comments, but they must use the CSS comment syntax that you've seen already:

```
/* This is the end of the section */
```

EXERCISE 14-6. Playing with pseudo-elements

In this exercise, we'll add some generated content to the list items in the "Entertainment" section of the *specials.html* document.

1. Start by inserting a little star icon (*star-icon.png*) before every list item by adding this style rule to the style sheet:

   ```
   #entertainment li::before {
     content: url(images/star-icon.png);
     margin-right: 1em;
   }
   ```

   The **`margin-right`** property adds a little space between the image and the text (you'll learn more about margin properties in **Chapter 15**). Save the file and open it in your browser to make sure the stars are there before every item.

2. Two of the items in the list have the class "fave". We can insert text after those items with the **`::after`** pseudo-element:

   ```
   #entertainment li.fave::after {
     content: " (Fan Favorite)";
     color: darkolivegreen;
   }
   ```

   Notice that there is a character space before the parentheses in the content value. Without it, there would be no space between the marked-up and generated content. You can style that text however you like. I made it green, but you could also make it all uppercase or apply any other properties that are relevant to text.

   At this point, the list should look like the one in FIGURE 14-12.

3. Finally, try your hand at formatting the first letter of the paragraphs in the "This Month Only" section of the page using the **`::first-letter`** pseudo-element. I've made them a little larger, bold, sans-serif, and "darkolivegreen", but you can play with whatever styles you like:

   ```
   #monthly p::first-letter{
     font-size: 1.25em;
     font-weight: bold;
     font-family: sans-serif;
     color: darkolivegreen;
   }
   ```

And that's a wrap on pseudo-elements!

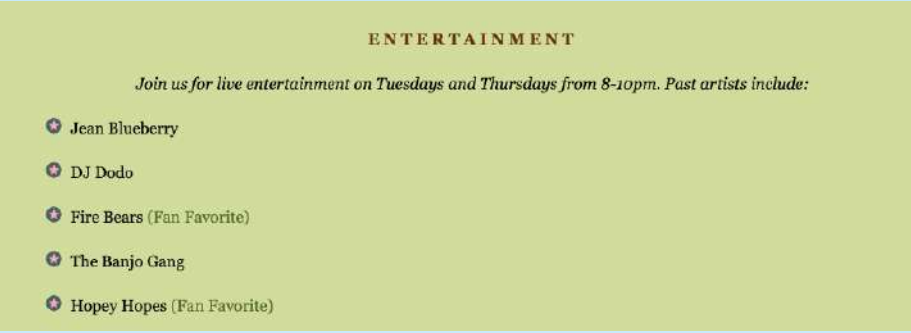

ENTERTAINMENT
Join us for live entertainment on Tuesdays and Thursdays from 8-10pm. Past artists include:
Jean Blueberry
DJ Dodo
Fire Bears (Fan Favorite)
The Banjo Gang
Hopey Hopes (Fan Favorite)

FIGURE 14-12. The resulting "Entertainment" section after playing around with pseudo-elements

The style sheet should be named with the *.css* suffix (there are some exceptions to this rule, but you're unlikely to encounter them as a beginner). It may begin with the **@charset** at-rule to declare the character encoding, although you need to do that only if you are using an encoding other than UTF-8. If you use **@charset**, it must be the first element in the style sheet, with no characters, including comments or style rules, preceding it.

FIGURE 14-13 shows how a style sheet document looks in my code editor.

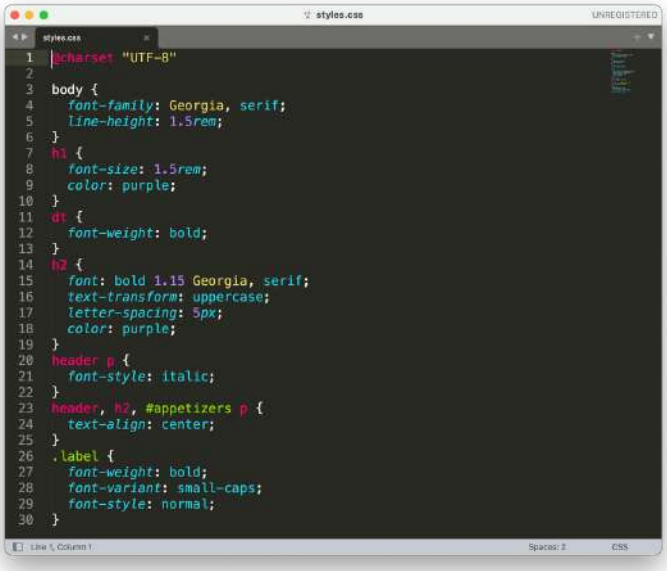


FIGURE 14-13. External style sheets contain only CSS rules and comments.

There are two ways to apply an external style sheet: an **@import** rule and the **link** element. Let's look at both of these attachment methods.

Importing with @import

One method for attaching an external style sheet to a document is to import it with an **@import** rule. The **@import** at-rule is another type of rule you can add to a style sheet, either in an external *.css* style sheet document or right in the **style** element, as shown in the following example:

```
<head>
  <style>
     @import url("/path/stylesheet.css");
     /* other styles */
  </style>
  <title>Titles are required.</title>
</head>
```

In this example, a relative URL is shown, but it could also be an absolute URL (beginning with **https://**). The **@import** rule must go at the beginning of the

style sheet, *before any selectors*, but after **@charset** if it is used. You can import more than one style sheet, and they all will apply, but rules from the last style sheet listed take precedence over earlier ones when there are conflicts.

You can also limit a style sheet's import to specific media types (**screen**, **print**, or **all**) or display features (orientation, screen size, etc.) using **media queries**. Media queries are a method for applying styles based on the medium used to display the document. They appear after the **@import** rule. For example, if you have created a style sheet that should be imported and used only when the document is printed, use this rule:

```
@import url(print_styles.css) print;
```

Or, to serve a special style sheet just for small devices, you could query the viewport:

```
@import url(small_device.css) screen and (max-width: 420px;);
```

I'll talk a lot more about media queries in **Chapter 19**, but I mention them here as they are relevant to importing style sheets.

Using the link Element

The **link** element defines a relationship between the current document and an external resource. You've already seen it in use in **Chapter 12** when we used it to link to web fonts. **link** is also commonly used to link to external style sheets. The **link** element goes in the **head** of the document, as shown here:

```
<head>
  <title>Titles are required.</title>
  <link rel="stylesheet" href="/path/stylesheet.css">
</head>
```

You need to include two attributes in the **link** element (see **Note**):

rel="stylesheet"

Defines the linked document's relation to the current document. The value of the **rel** attribute is always **stylesheet** when you are linking to a style sheet.

href="*url*"

Provides the location of the *.css* file.

You can include multiple **link** elements to different style sheets, and they'll all apply. If there are conflicts, whichever one is listed last will override previous settings, because of the rule order and the cascade.

You can apply a style sheet only to a particular medium (as you just saw for the **@import** rule) by specifying the media type with the **media** attribute (see **Note**). Here is a print-specific style sheet added with **link**:

```
<link rel="stylesheet" href="/path/stylesheet.css" media="print">
```

You can try out the **link** and **@import** methods in EXERCISE 14-7.

NOTE

The **link** *element may also include a* **type** *attribute to identify the file type of the linked resource, but when linking .css documents, it is recommended that you omit the* **type="text/css"**. *The* **type** *attribute is no longer required for style sheets.*

NOTE

While targeting specific media in a **link** *element is an option, the preferred method is to handle media types in the style sheets themselves.*

EXERCISE 14-7. Making an external style sheet

It's OK to use an embedded style sheet while working out the design of a single page, as we've been doing in the exercises so far. But once the design is finished, saving the styles in an external style sheet allows them to be shared by multiple pages on the site. Let's do that for the style sheet we've been working on throughout this chapter.

1. Open the latest version of *specials.html*. Our style sheet has gotten pretty long! Select and cut all of the rules within the **style** element, but leave the opening and closing **<style>** tags, because we'll be using them in a moment.
2. Create a new plain ASCII text document in a text editor (or a new document in a code editor), and paste in all of the style rules. Make sure that no markup got in there by accident.
3. Save this document as *specials-styles.css* in the same directory as the *specials.html* document.
4. Now, back in *specials.html*, add an **@import** rule to attach the external style sheet:

```
<style>
  @import url(specials-styles.
css);
</style>
```

   Save the file and reload it in the browser. It should look exactly the same as it did when the style sheet was embedded. If not, go back and make sure that no rules were omitted.

5. Delete the whole **style** element, and this time add the style sheet with a **link** element in the **head** of the document:

```
<head>
  <link rel="stylesheet"
href="specials-styles.css">
  ...
```

Again, test your work by saving the document and taking a look at it in the browser. Hopefully, nothing has changed.

Using Modular Style Sheets

Because browsers can compile information from multiple external style sheets, modular style sheets have become a popular technique for style management. Many developers keep styles they frequently reuse—such as typography treatments, layout rules, or form-related styles—in separate style sheets, then combine them in a mix-and-match fashion using **@import** rules. Again, the **@import** rules need to go before rules that use selectors.

Here's an example of a style sheet that imports multiple external style sheets:

```
/* basic typography */
@import url("type.css");

/* form inputs */
@import url("forms.css");

/* navigation */
@import url("list-nav.css");

/* site-specific styles */
body { background: orange; }

/* more style rules */
```

NOTE

You can also supply the URL without the **url()** *notation:*

```
@import "/path/style.css";
```

Again, absolute pathnames, beginning at the root, will ensure that the .css document will always be found.

This is a good technique to keep in mind as you gain experience creating sites. You'll find that there are some style solutions that work well for you, and it's nice not to have to reinvent the wheel for every new site.

Modular style sheets are a good time-saving and organizational device; however, they can be problematic for performance and caching. If you use this method, it is recommended that you compile all of the styles into a single document before delivering them to a browser, to improve performance. Not to worry, you don't need to do this manually; there are tools out there that will do it for you. One example is Lightning CSS (***lightningcss.dev***) which is a style sheet bundler that works with Webpack, Vite, Node.js, and other build tools. PostCSS (***postcss.org***) is another popular tool that can compile *.css* files.

WRAPPING IT UP

In this important chapter, you got to round out your collection of CSS selectors. You should be comfortable selecting elements by their attributes and values. You've learned a good method for styling links in a way that provides valuable feedback, using a stack of pseudo-class selectors. You've seen how

you can select elements based on their position in the DOM structure relative to their siblings and parent elements, and you know how to exclude elements from your selection. The pseudo-elements let you style the first letter or first line of an element or even add content before or after an element when the browser renders the page. Finally, you've learned the benefits of putting your styles in an external style sheet and how to bring it into your document with `<link>` and `@import`.

TEST YOURSELF

See how much you remember about selectors with this little quiz. Answers are in **Appendix A**.

1. Which of the following selectors target the highlighted (orange) element only?

```
<div class="container">
  <h1>The Beginning</h1>
  <p>Hello, world!</p>
</div>
```

   a. `.container p { }`

   b. `.container > p { }`

   c. `div p { }`

   d. All of the above

2. Which selector does *not* target the highlighted element?

```
<div class="container">
  <div class="wrapper">
    <p>Hello, world!</p>
  </div>
</div>
```

   a. `.container p { }`

   b. `.container > p { }`

   c. `.container .wrapper p { }`

   d. `.container * { }`

3. Which selector targets the highlighted element?

```
<div class="container">
  <div id="wrapper">
    <p>Hello, world!</p>
  </div>
</div>
```

   a. `.wrapper { }`

   b. `div ~ wrapper { }`

   c. `@wrapper { }`

   d. `#wrapper { }`

4. Which of the following selectors target the following element?

```
<a href="#" class="btn">Click me</a>
```

   a. `.btn { }`
   b. `a[href="#"] { }`
   c. Both of the above
   d. None of the above

5. Match each pseudo-class with the element(s) it targets.

a. _____	`a:link`	1. Links that have already been clicked
b. _____	`a:visited`	2. An element that is highlighted and ready for input
c. _____	`a:hover`	3. A link in the process of being clicked or tapped
d. _____	`a:active`	4. A link with the mouse pointer over it
e. _____	`:focus`	5. Links that have not yet been visited

6. Which of the following selectors target the highlighted element?

```
<ul id="seeds">
  <li><img src="flowers-yellow.jpg" alt="Yellow flowers"
class="thumbnail"></li>
  <li><img src="flowers-red.jpg" alt="Red flowers"
class="thumbnail"></li>
  <li><img src="flowers-blue.jpg" alt="Blue flowers"
class="thumbnail"></li>
</ul>
```

   a. `img[src$="-red.jpg"] { }`
   b. `img[src^="-red.jpg"] { }`
   c. `img[src*="-red.jpg"] { }`
   d. a and c

7. Which selector does *not* target the highlighted **`img`** elements?

```
<ul id="seeds">
  <li><img src="flowers-yellow.jpg" alt="Yellow flowers"
class="thumbnail"></li>
  <li><img src="flowers-red.jpg" alt="Red flowers"
class="thumbnail"></li>
  <li><img src="flowers-blue.jpg" alt="Blue flowers"
class="thumbnail"></li>
</ul>
```

   a. `#seeds li img { }`
   b. `img.class="thumbnail" { }`
   c. `*[alt~="flowers"] { }`
   d. `img.thumbnail { }`

8. Which of the following selectors target the highlighted `li` element?

```
<ul id="flowers">
  <li>Petunia</li>
  <li>Dahlia</li>
  <li>Hyacinth</li>
  <li>Marigold</li>
  <li>Daisy</li>
  <li>Poppy</li>
</ul>
```

   a. `#flowers li:nth-of-type(3) { }`

   b. `#flowers li:nth-child(3) { }`

   c. `#flowers li:nth-last-child(4) { }`

   d. All of the above

9. Which of the following selectors do *not* target the highlighted `li` elements?

```
<ul id="flowers">
  <li>Petunia</li>
  <li>Dahlia</li>
  <li>Hyacinth</li>
  <li>Marigold</li>
  <li>Daisy</li>
  <li>Poppy</li>
</ul>
```

   a. `li:nth-child(odd) { }`

   b. `li:nth-child(1), li:nth-child(3), li:nth-child(5) { }`

   c. `li:nth-child(1, 3, 5) { }`

   d. `li:nth-last-of-type(2n+2) { }`

   e. c and d

10. Write the style rule that would display the following markup, as shown in FIGURE 14-14. Remember, you need to provide the added content and style it.

```
<p class="quote">Done is beautiful!</p>
```

QUOTE OF THE DAY: Done is beautiful!

FIGURE 14-14. The code example from question 10 as displayed in a browser

CHAPTER 15

THINKING INSIDE THE BOX

THE CSS BOX MODEL

In **Chapter 11, Introducing Cascading Style Sheets**, I described the **box model** as one of the fundamental concepts of CSS. According to the box model, every element in a document generates a box to which properties such as width, height, padding, borders, and margins can be applied. You probably have a feel for how element boxes work from adding backgrounds to elements. This chapter covers all the box-related properties, starting with an overview of the components of an element box and then taking on the box properties from the inside out: content dimensions, padding, borders, and margins.

IN THIS CHAPTER

The parts of an element box

Setting box dimensions

Padding

Borders

Outlines

Margins

Assigning display roles

Adding a drop shadow

THE ELEMENT BOX

As you've seen, every element in a document, both block-level and inline, generates a rectangular **element box**. The components of an element box are diagrammed in FIGURE 15-1. Pay attention to the new terminology—it will be helpful for keeping things straight later in the chapter.

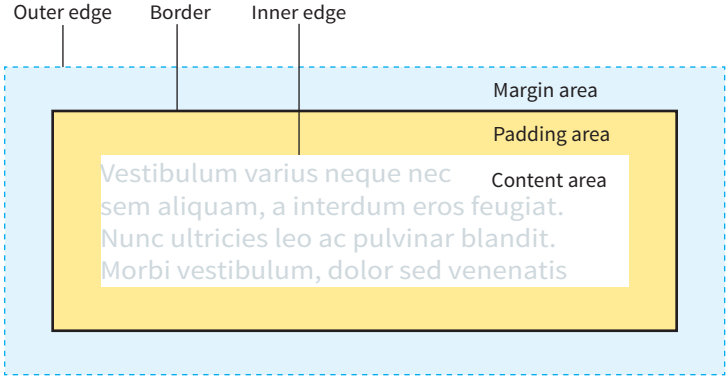

Outer edge
Border
Inner edge
Margin area
Padding area
Content area
Vestibulum varius neque nec sem aliquam, a interdum eros feugiat. Nunc ultricies leo ac pulvinar blandit. Morbi vestibulum, dolor sed venenatis

FIGURE 15-1. The parts of an element box according to the CSS box model

The amount of space an element takes up on the page includes the content plus the padding, borders, and margins applied to the element.

The defined aspects of an element box are as follows:

Content area

At the core of the element box is the content itself. In FIGURE 15-1, the content area is indicated by a white box.

Inner edges

The edges of the content area are referred to as the inner edges of the element box. Although the inner edges are made distinct by a color change in FIGURE 15-1, in real pages, the edges of the content area are invisible.

Padding

The padding is the area between the content area and an optional border. In the diagram, the padding area is indicated in yellow. Padding is optional.

Border

The border is a line (or stylized line) that surrounds the element and its padding. Borders are also optional.

Margin

The margin is an optional amount of space added on the *outside* of the border. In the diagram, the margin is indicated with light-blue shading, but in reality, margins are always transparent, allowing the background of the parent element to show through.

Outer edge

The outer edge of the element box corresponds to the outside edge of the margin area. In other words, the total amount of space an element takes up on the page includes the content plus any padding, borders, and margins applied to the element. The outer edge in the diagram is indicated with a dotted line around the margin area, but in real web pages, the edge of the margin is invisible.

All elements have these box components; however, as you will see, some properties behave differently based on whether the element displays as block or inline. In fact, we'll see some of those differences right away as we look at box dimensions.

SPECIFYING BOX DIMENSIONS

`width`, `inline-size`

Values: *length* | *percentage* | *auto*

Default: `auto`

Applies to: Block-level elements and replaced inline elements (such as images)

Inherits: No

height, block-size

Values: *length* | *percentage* | *auto*

Default: auto

Applies to: Block-level elements and replaced inline elements (such as images)

Inherits: No

box-sizing

Values: *content-box* | *border-box*

Default: content-box

Applies to: All elements

Inherits: No

By default, the width and height of a block element are calculated automatically by the browser, thus the default **auto** value. The box will be as wide as the block element that contains the element (its containing element) or the viewport (browser window), and as tall as necessary to fit the content. However, you can use the **width** and **height** properties, or their logical equivalents **inline-size** and **block-size** (see the **"Logical Properties and Values"** sidebar), to make the content area of an element a specific width or height.

Box Sizing

Unfortunately, setting box dimensions is not as simple as just dropping those properties in your style sheet. You have to know exactly which part of the element box you are sizing.

Logical Properties and Values

The W3C has been putting a lot of effort into making CSS more friendly to all languages, including those written in directions other than from left to right. As part of that effort, they have introduced new "logical" terminology for concepts such as "width" and "left/right" to define certain properties and values contextually based on the writing direction of the document.

Logical (or "flow-relative") properties use the terms "inline" and "block" to establish a direction. "Inline" refers to the writing direction of the document, and "block" is the direction perpendicular to inline. For example, in English, the inline direction is horizontal because we write in lines from left to right, and the block direction is vertical (lines go from top to bottom). In Mongolian, which is written vertically, the inline direction is vertical, and the block direction is horizontal.

With inline and block directions as a basis, many existing CSS properties and values have been recast to work in any language. You've already seen how **width** can be expressed as **inline-size** and **height** as **block-size**. In addition, physical and directional terms like top, bottom, left, and right have **block-start**, **block-end**, **inline-start**, and **inline-end** equivalents.

You will see the logical terminology in properties and values in action throughout this chapter as we discuss padding, borders, and margins, and again when we talk about using Flexbox and Grid Layout later in the book.

Browser support

Logical properties have been supported since 2021 and are at around 97% saturation as of this writing. If your site requires that you support very old browsers and Internet Explorer, you may want to stick with the traditional properties instead.

Additional resources

- "CSS Logical Properties and Values" by Ollie Williams (*css-tricks.com/css-logical-properties-and-values/*)
- CSS Logical Properties and Values Level 1 (*w3.org/TR/css-logical-1/*)
- CSS Writing Modes Level 3 (*w3.org/TR/css-writing-modes-3/*)

The **box-sizing** property allows us to choose from two approaches to element sizing. The default method (**content-box**) applies the width and height values to the *content box* (no surprise there). That means that the resulting size of the **visible element box** will be the dimensions you specify *plus* any padding and borders that have been added to the element. The other method (**border-box**) applies the width and height values to the *border box*, which includes the content as well as the padding and border. With this method, the resulting visible element box, including padding and borders, will be exactly the dimensions you specify.

Regardless of the method you choose, you can specify the width and height only for block-level elements and nontext inline elements, such as images. The width- and height-related properties do not apply to inline text (**non-replaced**) elements and are ignored by the browser. In other words, you cannot specify the width and height of an anchor (**a**) or **strong** element (see **Note**).

NOTE

Actually, there is a workaround for applying **width** *and* **height** *properties to inline elements such as anchors (***a***): you can force them to behave as block elements with the* **display** *property, which I'll cover at the end of this chapter.*

Knowing the resulting size of your elements is critical to getting layouts to behave predictably. In the following sections, you'll see how the **box-sizing** values make a big impact on how sizes are calculated.

Sizing the Content Box

By default (that is, if you do not include a **box-sizing** rule in your styles), the **width** and **height** properties are applied to the content box (see **Note**). That is the way all current browsers interpret width and height values, but you can explicitly specify this behavior by setting **box-sizing: content-box**.

NOTE

To keep the explanations and examples throughout this chapter easier to read, I'll be focusing on the traditional, physical property names, but keep in mind that you also have the option of using their logical equivalents. In some instances, which I'll be sure to point out, doing so even offers an advantage. You will find the logical properties included in the property listings for each property type.

In the following example, a simple paragraph is given a width of 500 pixels and a height of 150 pixels, with 20 pixels of padding, a 5-pixel border, and a 20-pixel margin all around (I'm sticking with pixel values here because the math will be more evident). In the default content box model, the **width** and **height** values are applied to the *content area only*:

```
p {
  width: 500px;
  height: 150px;
  padding: 20px;
  border: 5px solid gray;
  margin: 20px;
  background: #f2f5d5;
}
```

As you can see in **FIGURE 15-2**, the resulting width of the *visible* element box ends up being 550 pixels—the content area plus 40px padding (20px left and right) and 10px of border (5px left and right):

Visible element box =

5px + 20px + 500px width + 20px + 5px = 550 pixels

When you throw in 40 pixels of margin, the width of the *entire* element box is 590 pixels:

Element box =

20px + 5px + 20px + `500px width` + 20px + 5px + 20px = 590 pixels

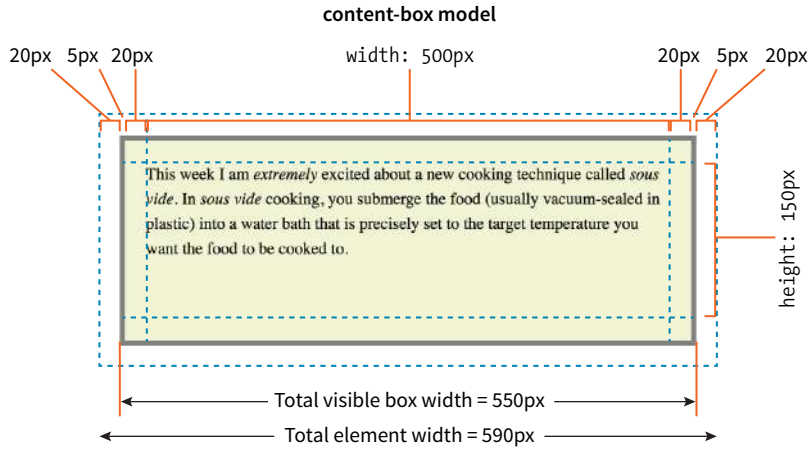

content-box model
20px 5px 20px
width: 500px
20px 5px 20px
This week I am extremely excited about a new cooking technique called sous vide. In sous vide cooking, you submerge the food (usually vacuum-sealed in plastic) into a water bath that is precisely set to the target temperature you want the food to be cooked to.
height: 150px
Total visible box width = 550px
Total element width = 590px

FIGURE 15-2. Specifying the `width` and `height` with the `content-box` model

Using the border-box Model

The other way to specify the size of an element is to apply width and height dimensions to the entire visible box, including the padding and border. Because this is not the default browser behavior, to do this you need to explicitly set **`box-sizing: border-box`** in the style sheet.

Let's look at the same paragraph example from the previous section and see what happens when we make it 500 pixels wide using the **`border-box`** method. All other style declarations for the box stay the same:

```
p {
  box-sizing: border-box;
  width: 500px;
  height: 150px;
  padding: 20px;
  border: 5px solid gray;
  margin: 20px;
  background: #f2f5d5;
}
```

As FIGURE 15-3 shows, now the width of the visible box is 500 pixels (compared to 550 pixels in the **`content-box`** model), and the total element width is 540px.

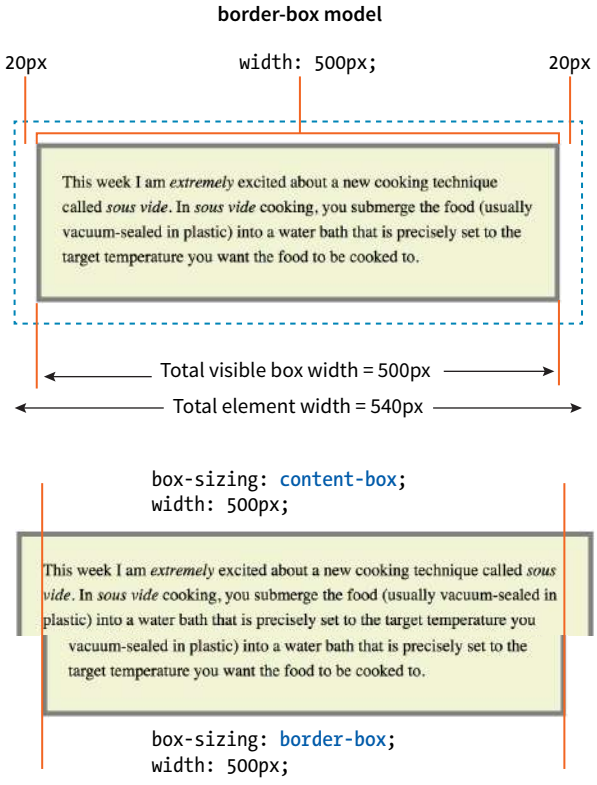

border-box model
20px
width: 500px;
20px
This week I am extremely excited about a new cooking technique called sous vide. In sous vide cooking, you submerge the food (usually vacuum-sealed in plastic) into a water bath that is precisely set to the target temperature you want the food to be cooked to.
Total visible box width = 500px
Total element width = 540px
box-sizing: content-box;
width: 500px;
This week I am extremely excited about a new cooking technique called sous vide. In sous vide cooking, you submerge the food (usually vacuum-sealed in plastic) into a water bath that is precisely set to the target temperature you
vacuum-sealed in plastic) into a water bath that is precisely set to the target temperature you want the food to be cooked to.
box-sizing: border-box;
width: 500px;

FIGURE 15-3. Sizing an element with the `border-box` method. The bottom diagram compares the resulting boxes from the two sizing methods.

Many developers find the **`border-box`** model to be a more intuitive way to size elements. It's particularly helpful for specifying widths in percentages, which is a cornerstone of responsive design. For example, you can make two columns each 50% wide and know that they will fit next to each other in the available space without you needing to mess around with adding padding and border widths to the mix (although you still need to account for margins).

In fact, it is standard practice simply to set *everything* in the document to use the **`border-box`** model by setting the **`box-sizing`** property on the root (**`html`**) element, then setting all other elements to inherit it using the universal selector (**`*`**), like this:

```
html {box-sizing: border-box;}

*, *:before, *:after {box-sizing: inherit;}
```

This is a best practice particularly if you are working with a site building platform that pulls together various page components that may have their own **`box-sizing`** settings. For more information on this technique, read Chris Coyier's article "Inheriting box-sizing Probably Slightly Better Best-Practice" (*oreil.ly/FMmG7*).

Specifying Height

The **height** property works just the same as **width**, but in general practice it's less common to specify the height of elements. It is more in keeping with the nature of the medium to allow the height of the element box to be calculated automatically, allowing it to change based on the font size, user settings, or other factors. If you do specify a height for an element containing text, be sure to also consider what happens should the content not fit. Fortunately, CSS gives you some options, as you'll see in the next section.

Handling Overflow

When an element box is sized too small for its contents, you can specify what to do with the content that doesn't fit with the **overflow** property.

`overflow`

Values: `visible` | `hidden` | `scroll` | `auto`

Default: `visible`

Applies to: Block-level elements and replaced inline elements (such as images)

Inherits: No

FIGURE 15-4 demonstrates the predefined values for **overflow**. In the figure, the various values are applied to an element that is 150 pixels square. The background color makes the edges of the content area apparent.

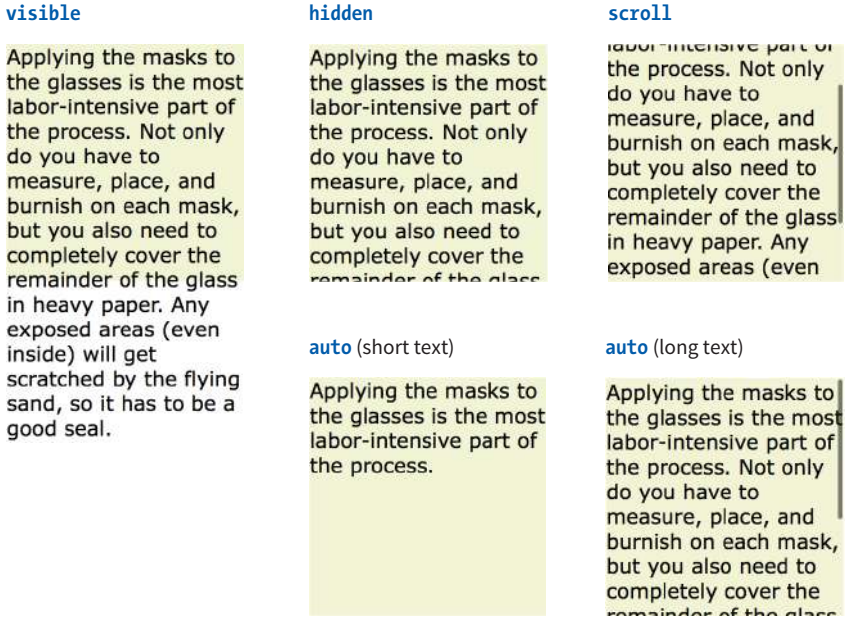

visible
Applying the masks to the glasses is the most labor-intensive part of the process. Not only do you have to measure, place, and burnish on each mask, but you also need to completely cover the remainder of the glass in heavy paper. Any exposed areas (even inside) will get scratched by the flying sand, so it has to be a good seal.
hidden
Applying the masks to the glasses is the most labor-intensive part of the process. Not only do you have to measure, place, and burnish on each mask, but you also need to completely cover the
scroll
the process. Not only do you have to measure, place, and burnish on each mask, but you also need to completely cover the remainder of the glass in heavy paper. Any exposed areas (even
auto (short text)
Applying the masks to the glasses is the most labor-intensive part of the process.
auto (long text)
Applying the masks to the glasses is the most labor-intensive part of the process. Not only do you have to measure, place, and burnish on each mask, but you also need to completely cover the

FIGURE 15-4. Options for handling content overflow. The **scroll** and **auto** options have narrow gray scrollbars to the right of the text (as rendered on macOS).

Maximum and Minimum Dimensions

If you want to set a limit on the size of a block element, use the **max-** and **min-** prefixed properties.

`max-height`,
`max-width`,
`max-block-size`,
`max-inline-size`

Values:
length | *percentage* | none | `max-content` | `min-content` | `fit-content` | `fit-content()`

`min-height`,
`min-width`,
`min-block-size`,
`min-inline-size`

Values:
length | *percentage* | `max-content` | `min-content` | `fit-content` | `fit-content()`

These properties work with block-level and replaced elements (like images) only. When the **content-box** model is used, the value applies to the content area only, so if you apply padding, borders, or margins, it will make the overall element box larger, even if a **max-width** or **max-height** property has been specified.

The possible values are:

`visible`

The default value, which allows the content to hang out over the element box so that all of it can be seen.

`hidden`

When `overflow` is set to `hidden`, the content that does not fit gets clipped off and does not appear beyond the edges of the element's content area.

`scroll`

When `scroll` is specified, scrollbars are added to the element box to let users scroll through the content. Be aware that they may become visible only when you click the element to scroll it.

`auto`

The `auto` value allows the browser to decide how to handle overflow. In most cases, scrollbars are added when the content doesn't fit and they are needed.

PADDING

Padding is the space between the content and the border.

Padding is the space between the content area and the border (or the place the border would be if one isn't specified). It is particularly helpful to add padding to elements when using a background color or a border. This gives the content a little breathing room and prevents the border or edge of the background from bumping right up against the text.

You can add padding to the individual sides of any element (block-level or inline). There is also a shorthand `padding` property that lets you add padding on all sides at once.

`padding-top`, `padding-right`, `padding-bottom`, `padding-left`, `padding-block-start`, `padding-inline-end`, `padding-block-end`, `padding-inline-start`

Values:	*length* | *percentage*
Default:	0
Applies to:	All elements
Inherits:	No

`padding`

Values:	*length* | *percentage*
Default:	0
Applies to:	All elements
Inherits:	No

The **padding-top**, **padding-right**, **padding-bottom**, and **padding-left** properties and their logical equivalents specify an amount of padding for each side of an element, as shown in this example and FIGURE 15-5 (note that I've also added a background color to make the outer edges of the padding area apparent):

```
blockquote {
  padding-top: 30px;
  padding-right: 60px;
  padding-bottom: 30px;
  padding-left: 60px;
  background-color: #D098D4; /* light green */
}
```

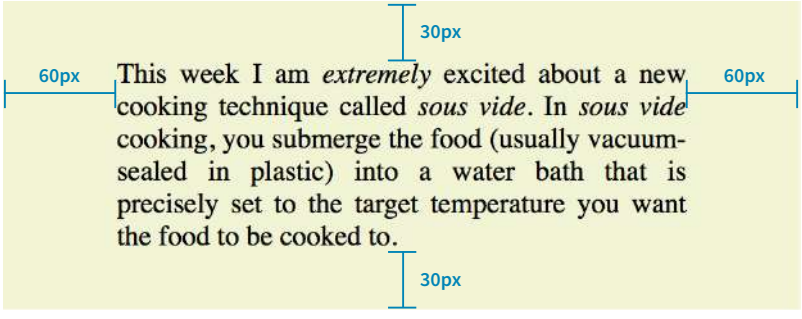

30px
60px
This week I am extremely excited about a new cooking technique called sous vide. In sous vide cooking, you submerge the food (usually vacuum-sealed in plastic) into a water bath that is precisely set to the target temperature you want the food to be cooked to.
60px
30px

FIGURE 15-5. Adding padding around the content of an element

You can specify padding in any of the CSS length units, such as **px**, **em**, or **rem**, or as a percentage of the width of the parent element—yes, the parent's *width* is used as the basis, even for top and bottom padding! If the width of the parent element changes, so will the padding values on all sides of the child element, which makes percentage values somewhat tricky to manage. You should also be aware that if you use rems or ems, the padding will be relative to the font size, which may result in too much padding if the user sets their browser fonts very large. When specifying padding, be sure to consider whether the padding space should be fixed or flexible, and choose the most appropriate unit for the job based on the design.

The Shorthand padding Property

As an alternative to setting padding one side at a time, you can use the shorthand **padding** property to add padding all around the element. The syntax is interesting; you can specify four, three, two, or just one value for a single **padding** property. Let's see how that works, starting with four values.

When you supply four **padding** values, they are applied to each side in *clockwise* order, starting at the top. Some people use the mnemonic device "TRouBLe" for the order *Top Right Bottom Left*. This is a common syntax for applying shorthand values in CSS, so take a careful look:

```
padding: top right bottom left;
```

Using the **padding** property, we could reproduce the padding specified with the four individual properties in the previous example like this:

```
blockquote {
  padding: 30px 60px 30px 60px;
  background-color: #D098D4;
}
```

If the left and right padding are the same, you can shorten the declaration by supplying only three values. The value for "right" (the second value in the string) will be mirrored and used for "left" as well. It's as though the browser assumes the "left" value is missing, so it just uses the "right" value on both sides. The syntax for three values is as follows:

```
padding: top right/left bottom;
```

This rule would be equivalent to the previous example because the padding on both the left and right edges of the element is set to 60px:

```
blockquote {
  padding: 30px 60px 30px;
  background-color: #D098D4;
}
```

Continuing with this pattern, if you provide only two values, the first one is used for the top and the bottom edges, and the second one is used for the left and right edges:

```
padding: top/bottom right/left;
```

Again, the same effect achieved by the previous two examples could be accomplished with this rule:

```
blockquote {
  padding: 30px 60px;
  background-color: #D098D4;
}
```

Note that all of the previous examples have the same visual effect as shown in FIGURE 15-5.

Finally, if you provide just one value, it will be applied to all four sides of the element. This declaration applies 15 pixels of padding on all sides of a **div** element:

```
div#announcement {
  padding: 15px;
  border: 1px solid;
}
```

Now it's time to get a feel for adding padding to elements in EXERCISE 15-1.

■ AT A GLANCE

Shorthand Values

1 value

`padding: 10px;`

Applied to all sides.

2 values

`padding: 10px 6px;`

First is top and bottom; second is left and right.

3 values

`padding: 10px 6px 4px;`

First is top; second is left and right; third is bottom.

4 values

`padding: 10px 6px 4px 10px;`

Applied clockwise to top, right, bottom, and left edges consecutively (TRBL).

Logical Shorthand Properties

Here's something logical properties offer that the shorthand **padding** property doesn't. If you want to add an amount padding to the left and right sides of an element at once (and your writing direction is horizontal), you can use the **padding-inline** property:

```
blockquote { padding-inline: 60px; }
```

Similarly, **padding-block** lets you specify an amount of padding to add to the top and bottom in one declaration:

```
h2 { padding-block: 30px; }
```

EXERCISE 15-1. Adding a little padding

In this exercise, we'll begin adding box properties to improve the appearance of the home page of the website for the fictional Black Goose Bakery. I've given you a head start by marking up the source (*bakery.html*). Unlike the pages we've worked with in previous exercises, the bakery page uses an external style sheet, *bakery-styles.css*. Everything we will be doing to format this page over the next few chapters happens in the CSS file, so you should never need to edit the HTML document; however, that is the file you will open in the browser to see the results of your style changes. All the files are available at *learningwebdesign.com/6e/materials*.

FIGURE 15-6 shows before and after shots of the home page. It's going to take several exercises over four chapters to get there, and padding is just the beginning. In **Chapter 17, Getting in Line with Flexbox**, we'll turn that ugly navigation list into a nice navigation menu bar (in the meantime, please avert your eyes), and we'll give the page a two-column layout suitable for larger screens in **Chapter 18, Grid Layout**.

Start by getting familiar with the source document. Open *bakery.html* in a browser and a text editor to see what you've got to work with. The style sheet has been added with an `@import` rule in the `style` element. The document has been marked up with `header` (including a `nav` section), `main`, `aside`, and `footer` sections.

Now take a look at *bakery-styles.css* in your text editor. I used comments in the style sheet to organize the styles related to each section (bonus points for you if you keep the styles organized as you go along). You will find styles for text formatting, colors, and backgrounds—all properties that we've covered so far in this book, so they should look familiar. Let's get started by adding some rules to *bakery-styles.css* to add padding to the elements.

NOTE

This design uses a Google web font called Stint. You will need to have an internet connection in order to see it. If you are working offline, you will see Georgia or some serif font instead, which is just fine for these purposes, but your page won't look exactly like the ones in the figures.

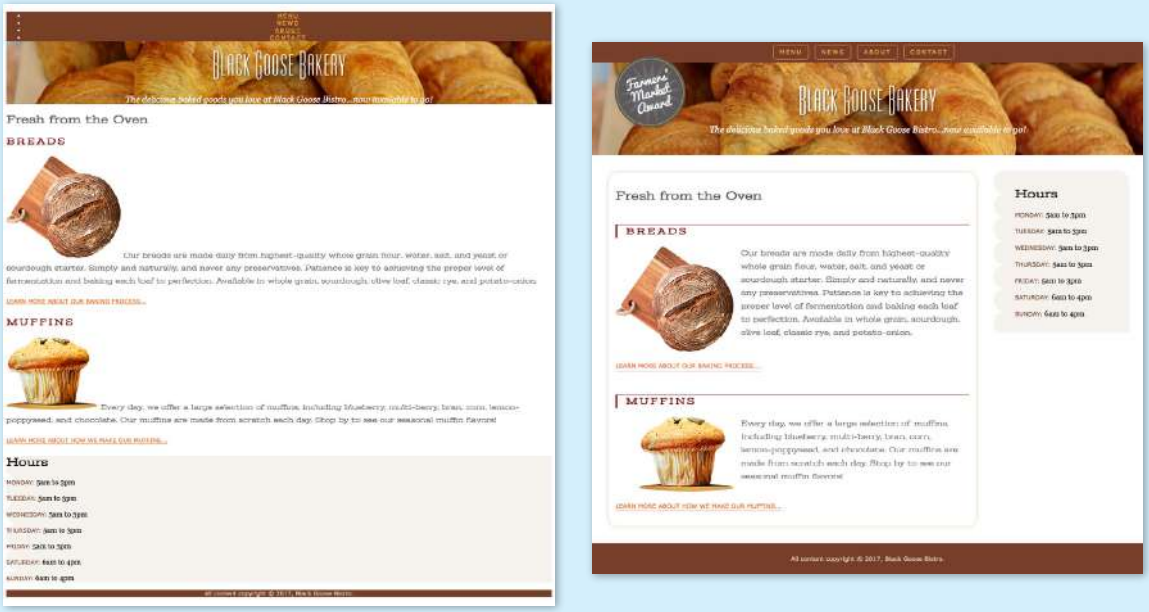


FIGURE 15-6. Before and after shots of the Black Goose Bakery site

EXERCISE 15-1. Continued

1. The first thing we'll do is set the **box-sizing** model to **border-box** for all the elements in the document. Add these new rules to the *bakery-styles.css*. This will make measurements simpler going forward:

```
html {
  box-sizing: border-box;
}
* {
  box-sizing: inherit;
}
```

2. Now find the styles for the **header** and give it a height. It will fill 100% of the width of the page by default, so the width is taken care of. I picked 240px for the height because it seemed tall enough to accommodate the content and show a nice amount of the croissant background image, but you can play around with it:

```
header {
  ...
  height: 240px;
}
```

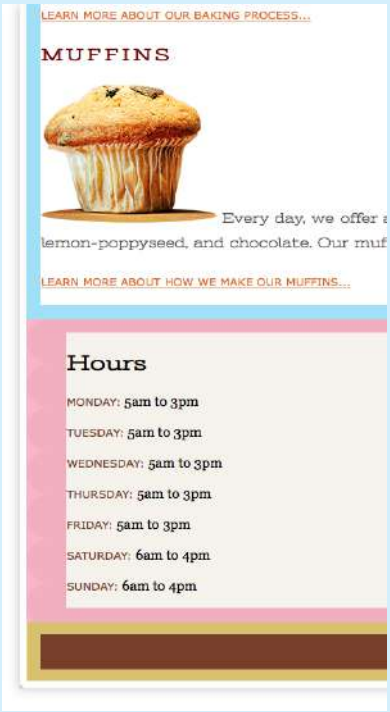

LEARN MORE ABOUT OUR BAKING PROCESS...
MUFFINS
Every day, we offer
lemon-poppyseed, and chocolate. Our muf
LEARN MORE ABOUT HOW WE MAKE OUR MUFFINS...
Hours
MONDAY: 5am to 3pm
TUESDAY: 5am to 3pm
WEDNESDAY: 5am to 3pm
THURSDAY: 5am to 3pm
FRIDAY: 5am to 3pm
SATURDAY: 6am to 4pm
SUNDAY: 6am to 4pm

FIGURE 15-7. The shaded areas indicate the padding added to **main** (blue), **aside** (pink), and **footer** (yellow). The colors were added to the figure for demo purposes but won't render in the browser.

3. The **main** section is going to need a little padding, so add 16px of padding on all sides. You can add this declaration after the existing **main** styles:

```
main {
  ...
  padding: 16px;
}
```

4. Next, we'll get a little fancier with the **aside** element ("Hours"). We'll need extra padding on the left side for the tiling scallop background image to be visible. There are several approaches to applying different padding amounts to each side, but we're going to do it in a way that gives you experience deliberately overriding earlier declarations.

   Use the **padding** shorthand property to add 16px of padding on all sides of the **aside** element. Then write a second declaration that adds 45px of padding on just the left side. Because the **padding-left** declaration comes second, it will override the 16px left side setting that was applied with the shorthand property:

```
aside {
  ...
  padding: 16px;
  padding-left: 45px;
}
```

5. Finally, that footer is looking skinny and cramped. Let's add some padding to that too, which will increase the height of the footer and give the content some space:

```
footer {
  ...
  padding: 12px;
}
```

6. Save the *bakery-styles.css* document, and then open (or reload) *bakery.html* in a browser to see the result of your work. The changes at this point are pretty subtle. FIGURE 15-7 uses colors to highlight the padding additions, but you will see only the added space.

BORDERS

A **border** is simply a line drawn around the content area and its (optional) padding. You can choose from eight border styles and make them any width and color you like. Borders can be applied all around the element or just on a particular side or sides. You can also give the box rounded corners. We'll start our border exploration with the various border styles.

Border Style

The **border-style** property is the most important of the border properties because, according to the CSS specification, if it is omitted, the border does not exist (the default is **none**). In other words, you must always declare the style of the border, or the other border properties will be ignored.

Border styles can be applied one side at a time or with the shorthand **border-style** property. The newer **border-block-style** and **border-inline-style** shorthand properties apply to just the top and bottom sides (block) and left and right sides (inline) of the element.

`border-top-style`, `border-right-style`,
`border-bottom-style`, `border-left-style`,
`border-block-start-style`, `border-inline-end-style`,
`border-block-end-style`, `border-inline-start-style`

Values:	none | solid | hidden | dotted | dashed | double | groove | ridge | inset | outset
Default:	none
Applies to:	All elements
Inherits:	No

`border-style`, `border-block-style`, `border-inline-style`

Values:	none | solid | hidden | dotted | dashed | double | groove | ridge | inset | outset
Default:	none
Applies to:	All elements
Inherits:	No

These properties each take as their value one of 10 keyword values describing the available border styles, which are illustrated in FIGURE 15-8. The value **hidden** is equivalent to **none**.

■ DESIGN TIP

Bottom Borders Instead of Underlines

Turning off link underlines and replacing them with a custom bottom border is a common design technique. It lightens the look of links while still making them stand out from ordinary text.

NOTE

The **border-block-style** *logical shorthand applies a style to the top and bottom sides.* **border-inline-style** *applies the border style to the left and right sides in horizontally written languages.*

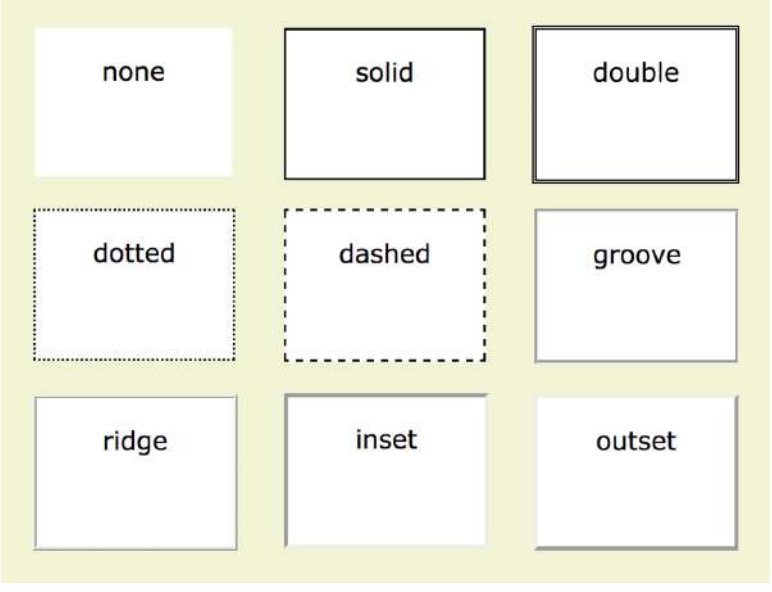


FIGURE 15-8. The available border styles (shown at the default medium width)

Use the side-specific border style properties (**border-top-style**, **border-right-style**, **border-bottom-style**, and **border-left-style** or their logical equivalents) to apply a style to one side of the element. If you do not specify a width, the default medium width will be used. If there is no color specified, the border uses the foreground color of the element (the same as the text).

In the following example, I've applied a different style to each side of an element to show the single-side border properties in action (FIGURE 15-9):

```
div#silly {
  border-top-style: solid;
  border-right-style: dashed;
  border-bottom-style: double;
  border-left-style: dotted;
  width: 300px;
  height: 100px;
}
```

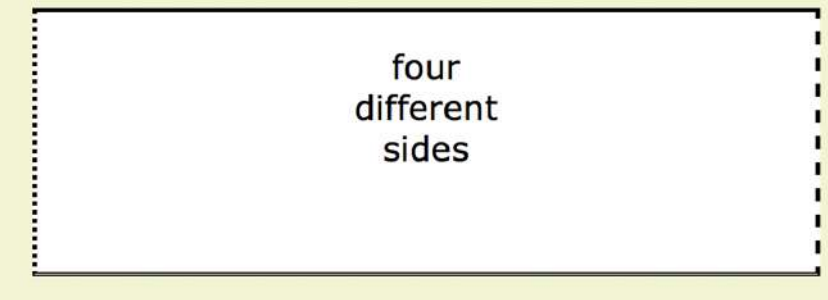


FIGURE 15-9. Border styles applied to individual sides of an element (clockwise from the top: solid, dashed, double, dotted)

The **border-style** shorthand property works on the clockwise (TRouBLe) system described for **padding** earlier. You can supply four values for all four sides, or fewer values when you want the left and right and/or top and bottom borders to be the same. The silly border effect in the previous example could also be specified with the **border-style** property as shown here, and the result would be the same as shown in FIGURE 15-9:

```
border-style: solid dashed double dotted;
```

Border Width (Thickness)

Use one of the **border-width** properties to specify the thickness of the border.

border-top-width, border-right-width,
border-bottom-width, border-left-width,
border-block-start-width, border-inline-end-width,
border-block-end-width, border-inline-start-width

Values: *length* | thin | medium | thick

Default: medium

Applies to: All elements

Inherits: No

border-width, border-block-width, border-inline-width

Values: *length* | thin | medium | thick

Default: medium

Applies to: All elements

Inherits: No

The most common way to specify the width of borders is using a pixel, rem, or em measurement; however, you can also specify one of the keywords (**thin**, **medium**, or **thick**) and leave the rendering up to the browser.

I've included a mix of width values in this example (FIGURE 15-10). Notice that I've also included the **border-style** property, because if I didn't, the border would not render at all:

```
div#thickness {
  border-top-width: thin;
  border-right-width: medium;
  border-bottom-width: thick;
  border-left-width: 12px;
  border-style: solid;
  width: 300px;
  height: 100px;
}
```

Once again, you can target each side of the element with a single-side property, or specify several sides at once in clockwise order with the shorthand **border-width** property. You can achieve the same effect by writing:

```
div#thickness {
  border-width: thin medium thick 12px;
  border-style: solid;
  width: 300px;
  height: 100px;
}
```

NOTE

The `border-block-width` *logical shorthand applies the border width to the top and bottom sides.* `border-inline-width` *applies the border width to the left and right sides in horizontally written languages.*

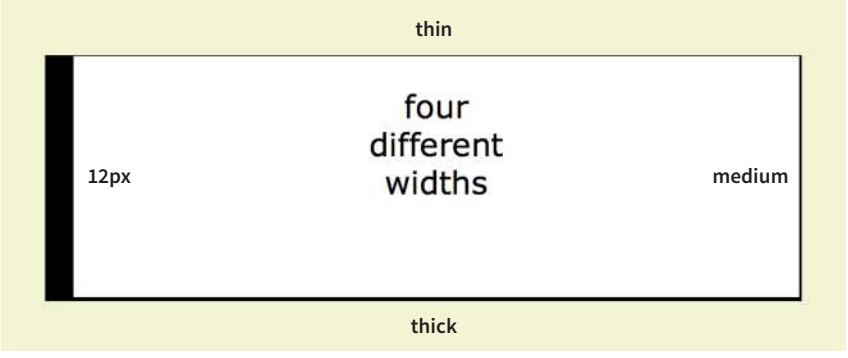


FIGURE 15-10. Specifying the width of borders

Border Color

Border colors are specified similarly, via the side-specific properties or the `border-color`, `border-block-color`, and `border-inline-color` shorthand properties. When you specify a border color, it overrides the foreground color as set by the `color` property for the element.

`border-top-color`, `border-right-color`, `border-bottom-color`, `border-left-color`, `border-block-start-color`, `border-inline-end-color`, `border-block-end-color`, `border-inline-start-color`

Values:	*color name* or *numeric value* | `transparent`
Default:	The value of the `color` property for the element
Applies to:	All elements
Inherits:	No

`border-color`, `border-block-color`, `border-inline-color`

Values:	*color name* or *numeric value* | `transparent`
Default:	The value of the `color` property for the element
Applies to:	All elements
Inherits:	No

You learned all about specifying color values in **Chapter 13, Colors and Backgrounds**, and you should be getting used to the shorthand properties as well, so I'll keep this example short and sweet. Here, I've provided two values for the shorthand `border-color` property to make the top and bottom of a `div` "orange" and the left and right sides "slategray" (FIGURE 15-11):

■ DESIGN TIP

Setting `border-color` to `transparent` allows the background to show through the border, yet holds the width of the border as specified. This can be useful when you're creating rollover (`:hover`) effects with borders, because the space where the border will appear is maintained even when the mouse is not over the element.

```
div#colorful {
  border-color: orange slategray;
  border-style: solid;
  border-width: 6px;
  width: 300px;
  height: 100px;
}
```

For the record, the logical shorthands could also be used to create the same (not very pretty) effect:

```
border-block-color: orange;
border-inline-color: slategray;
```

FIGURE 15-11. Specifying the color of borders

Combining Style, Width, and Color

The authors of CSS didn't skimp when it came to border shortcuts. They also created properties for providing style, width, and color values in one declaration, one side at a time. You can specify the appearance of specific sides, or use the **border** property to change all four sides at once.

`border-top`, `border-right`, `border-bottom`, `border-left`, `border-block-start`, `border-inline-end`, `border-block-end`, `border-inline-start`

Values: *border-style border-width border-color*

Default: Defaults for each property

Applies to: All elements

Inherits: No

`border`

Values: *border-style border-width border-color*

Default: Defaults for each property

Applies to: All elements

Inherits: No

The **border** and side-specific **border-*** properties may include style, width, and color values in any order. You do not need to declare all three, but if the border style value is omitted, no border will render.

The **border** shorthand property works a bit differently from the other shorthand properties that we've covered in that it takes one set of values and always applies them to all four sides of the element. In other words, it does not use the clockwise TRBL system that we've seen with other shorthand properties.

Here is a smattering of valid border shortcut examples to give you an idea of how they work:

```
h1 { border-left: red .5em solid; }  /* left border only */
h2 { border-block-end: 1px solid; }  /* bottom border only  */
p.example { border: 2px dotted #663; } /* all four sides */
```

Outlines are a good tool for checking your page layout as you work.

CSS Outlines

Another type of rule you can draw around an element is an **outline**. Outlines look like borders, and the syntax is the same, but there is an important difference. Outlines, unlike borders, are not taken into account when calculating the width of the element box. They just lie on top, not interfering with anything. Outlines are drawn on the outside edge of the border (if one is specified) and overlap the margin.

Because outlines do not affect layout, they're a great tool for checking your design. You can turn them on and off without affecting width measurements, to see where and how element boxes are positioned.

Avoid turning off outlines on any focusable elements (like links or buttons), as that creates accessibility challenges. A better way to prevent outlines from showing up when clicking is to use the **:focus-visible** pseudo-class, which applies only to elements that browsers highlight by default and ensures that users navigating with keyboards or other modalities are the only ones who see the outlines.

The outline properties are similar to the border properties, with one important difference: it is not possible to specify outlines for particular sides of the element box—it's all or nothing.

outline-style

Values: auto | solid | none | dotted | dashed | double | groove | ridge | inset | outset

Default: none

These are the same as the **border-style** values, with the addition of **auto**, which lets the browser choose the style. Also, you cannot set the **outline-style** to **hidden**.

outline-width

Values: *length* | thin | medium | thick

Default: medium

These are the same as the **border-width** values.

outline-color

Values: *color name* or *RGB/HSL value* | invert

Default: invert

The default **invert** value applies the inverse of the background color to the outline, but it has very little browser support.

outline-offset

Values: *length*

Default: 0

By default, the outline is drawn just outside the border edge. **outline-offset** moves the outline beyond the border by a specified length.

outline

Values: *outline-style outline-width outline-color*

Default: Defaults of individual properties

The shorthand **outline** property combines values for **outline-style**, **outline-width**, and **outline-color**. Remember that you can specify them only for all sides of the element at once—for example:

```
div#story { outline: 2px dashed red; }
```

Rounded Corners with border-radius

Perhaps you'd like your element boxes to look a little softer and rounder. Well, then, the **border-radius** property is for you! There are individual corner properties as well as a **border-radius** shorthand.

`border-top-left-radius`, `border-top-right-radius`, `border-bottom-right-radius`, `border-bottom-left-radius`, `border-start-start-radius`, `border-start-end-radius`, `border-end-end-radius`, `border-end-start-radius`

Values: *length | percentage*

Default: 0

Applies to: All elements

Inherits: No

`border-radius`

Values: *1, 2, 3, or 4 length or percentage values*

Default: 0

Applies to: All elements

Inherits: No

To round off the corners of an element, simply apply one of the **border-radius** properties—but keep in mind that you will see the result only if the element has a border or background color. Values are typically provided in pixels, rems, or ems.

You can target the corners individually or use the shorthand **border-radius** property. If you provide one value for **border-radius**, it is applied to all four corners. Four values are applied clockwise, starting in the top-left corner (top-left, top-right, bottom-right, bottom-left). When you supply two values, the first one is used for the top-left and bottom-right corners, and the second is used for the remaining two.

Compare the **border-radius** values to the resulting boxes in FIGURE 15-12. You can achieve many different effects, from slightly softened corners to a long capsule shape, depending on how you set the values relative to the size of the element.

Elliptical corners

So far, the corners we've made have been sections of perfect circles, but you can also make a corner elliptical by specifying two values, the first for the horizontal radius and the second for the vertical radius, like this (see FIGURE 15-13 **A** and **B**):

```
p {
  width: 200px;
  height: 100px;
  background: darkorange;
}
```

```
border-radius: 1em;
```

```
border-radius: 50px;
```

```
border-top-right-radius: 50px;
```

```
border-top-left-radius: 1em;
border-top-right-radius: 2em;
border-bottom-right-radius: 1em;
border-bottom-left: 2em;
```

or

```
border-radius: 1em 2em;
```

FIGURE 15-12. Applying different rounding styles to the corners of element boxes

A `border-top-right-radius: 100px 50px;`

B
```
border-top-right-radius: 50px 20px;
border-top-left-radius: 50px 20px;
```

If you want to use the shorthand property, separate the horizontal and vertical radii with a slash (otherwise, they'd be confused for different corner values). The following example sets the horizontal radius on all corners to 60px and the vertical radius to 40px (FIGURE 15-13 **C**):

C `border-radius: 60px / 40px;`

If you want to see something really nutty, take a look at a **`border-radius`** shorthand property that specifies a different ellipse for each of the four corners. All of the horizontal values are lined up on the left of the slash in clockwise order (top-left, top-right, bottom-right, bottom-left), and all of the corresponding vertical values are lined up on the right (FIGURE 15-13 **D**):

D `border-radius: 36px 40px 60px 20px / 12px 10px 30px 36px;`

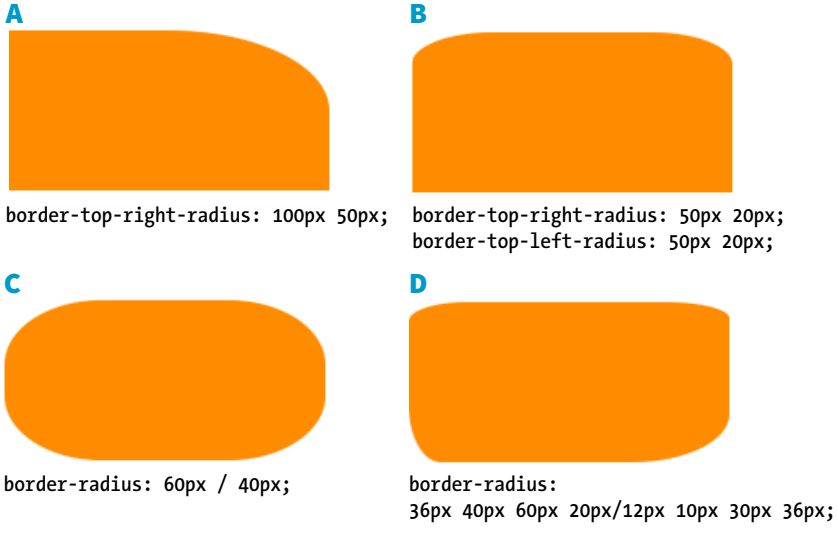


FIGURE 15-13. Applying elliptical corners to boxes

Now it's time to try your hand at borders. EXERCISE 15-2 will not only give you some practice, but should also give you some ideas about the ways borders can be used to add visual interest to designs.

EXERCISE 15-2. Border tricks

In this exercise, we'll have some fun with borders on the Black Goose Bakery page. In addition to putting borders around content sections of the page, we'll use borders to beef up the headlines and as an alternative to underlines under links.

1. Open *bakery-styles.css* in a text editor if it isn't already. We'll start with the basics by using the shorthand **border** property to add a tan double rule around the **main** element. Add the new declaration to the existing rule for **main**:

```
main {
  ...
  padding: 16px;
  border: double 6px #eaddc4;
}
```

2. Now try out some **border-radius** properties to add generous rounded corners to the **main** and **aside** sections. A 25-pixel radius should do. Pixels are my choice over ems here because I don't want the radius to scale with the text. Start by adding this declaration to the styles for **main**:

```
  border-radius: 25px;
```

   And give just the top-right corner of the **aside** element a matching rounded corner:

```
aside {
  ...
  border-top-right-radius: 25px;
}
```

3. Just for fun (and practice), we'll add a decorative border on two sides of the baked goods headings (**h3**). Find the existing rule for **h3** elements and add a declaration that adds a 1-pixel solid rule on the top of the headline. Add another that adds a thicker 3-pixel solid rule on the left. I want the borders to be the same color as the text, so I don't need to specify the **border-color**.

   Finally, to prevent the text from bumping into the left border, add a little bit of padding (1em) to the left of the headline content:

```
h3 {
  ...
  border-top: 1px solid;
  border-left: 3px solid;
  padding-left: 1em;
}
```

4. The last thing we'll do is replace the standard underline with a decorative bottom border under links. Start by turning off the underline for all links. Add this rule in the "link styles" section of the style sheet:

```
a {
  text-decoration: none;
}
```

   Then, add a 1-pixel dotted border to the bottom edge of links:

```
a {
  text-decoration: none;
  border-bottom: 1px dotted;
}
```

   As is often the case when you add a border to an element, it's a good idea to also add a little padding to give the content some breathing room:

```
a {
  text-decoration: none;
  border-bottom: 1px dotted;
  padding-bottom: .2em;
}
```

   Now you can save the style sheet and reload *bakery.html* in your browser. FIGURE 15-14 shows a detail view of how your page should be looking so far.

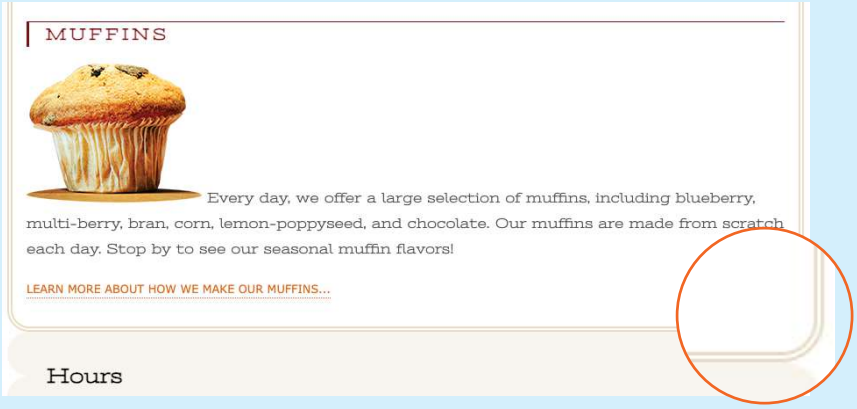

MUFFINS
Every day, we offer a large selection of muffins, including blueberry, multi-berry, bran, corn, lemon-poppyseed, and chocolate. Our muffins are made from scratch each day. Stop by to see our seasonal muffin flavors!
LEARN MORE ABOUT HOW WE MAKE OUR MUFFINS...
Hours

FIGURE 15-14. The results of our border additions

Picture-Perfect Borders

CSS Box Model Module Level 3 introduced the **border-image-*** properties, which let you fill in the sides and corners of a border box with an image of your choice, as shown in FIGURE 15-15.

Border images are applied with a collection of five properties:

- **border-image-source** indicates the location of the image.
- **border-image-slice** divides the image into nine sections using offset measurements.
- **border-image-width** specifies the width of the border area.
- **border-image-repeat** specifies whether the image should stretch or repeat along the sides.
- **border-image-outset** pushes the border away from the content by the specified amount.

There is also a shorthand **border-image** property that combines the individual properties in the following syntax:

```
border-image: source slice / width / outset
repeat;
```

The style rules for the image border in FIGURE 15-15 are as follows:

```
border: 5px solid #d1214a; /* red */
border-image: url(fancyframe.png) 55 fill / 55px /
25px stretch;
```

The **border** shorthand property provides a fallback style for the border should the image not load or if **border-image** isn't supported by the browser.

The **border-image** rule tells the browser to apply the image *fancyframe.png* to the border, slice it 55 pixels from the edges, and use the center of the image to fill the center of the box. The width of the border area is 55px, and the image should be pushed toward the margins by 25px. Finally, the image areas that make up the sides should stretch to fill the width and height of the box.

That's not much of an explanation, I know, but my article "Border Images" (available at *learningwebdesign.com/articles/*) goes into more detail. For even more information on border images, check out these resources:

- CSS Background and Borders Module Level 3 (*oreil.ly/dhaDy*)
- The Almanac entry for **border-image** on CSS-Tricks (*oreil.ly/7vQEt*), for a less dense explanation

55px
55px
55px
55px
Tapenade is a really simple dish to prepare and it's always a big hit at parties. Make this the night before so that the flavors have time to blend. Just bring it up to room temperature before you serve it. In the winter, try serving it warm.
Tapenade is a really simple dish to prepare and it's always a big hit at parties. Make this the night before so that the flavors have time to blend. Just bring it up to room temperature before you serve it. In the winter, try serving it warm.

fancyframe.png With border image Without border image

FIGURE 15-15. Examples of a border image applied to a box

MARGINS

A **margin** is an optional amount of space that you can add on the outside of the element's border box. Margins keep elements from bumping into one another or the edge of the browser window or viewport.

The side-specific and shorthand **margin** properties work much like the **padding** properties we've looked at already; however, margins have some special behaviors to be aware of.

`margin-top`, `margin-right`,
`margin-bottom`, `margin-left`,
`margin-block-start`, `margin-inline-end`,
`margin-block-end`, `margin-inline-start`

Values: *length* | *percentage* | `auto`

Default: `auto`

Applies to: All elements

Inherits: No

`margin`, `margin-block`, `margin-inline`

Values: *length* | *percentage* | `auto`

Default: `auto`

Applies to: All elements

Inherits: No

The margin properties are very straightforward to use. You can either specify an amount of margin to appear on each side of the element or use the **margin** property to specify all sides at once. **margin-block** assigns a margin to the top and bottom edges, and **margin-inline** applies to the left and right edges for horizontally written languages like English.

The shorthand **margin** property works the same as the **padding** shorthand. When you supply four values, they are applied in clockwise order (top, right, bottom, left) to the sides of the element. If you supply three values, the middle value applies to both the left and right sides. When two values are provided, the first is used for the top and bottom, and the second applies to the left and right edges. Finally, one value will be applied to all four sides of the element.

As with most web measurements, ems, rems, pixels, and percentages can be used to specify margins. Many developers and web design tools (such as Figma) use fixed pixel measurements for margins, because they are the most predictable. Using rem and em units has the advantage that the margins will be based on the font size, which is generally desirable. For example, rems are a good choice for adding space above and below paragraphs and headings, because you'd expect those spaces to stay proportional to the text. If you specify a percentage value, it is calculated based on the *width* of the parent

■ CSS TIP

Browser Default Margins

You may have noticed that space is added automatically around headings, paragraphs, and other block elements. That's the browser's default style sheet at work, applying margins above and below those elements.

It's good to keep in mind that the browser is applying its own values for margins behind the scenes. These values will be used unless you specifically override them with your own style rules.

If you are working on a design and coming across mysterious amounts of space that you didn't add, the browser's default styles may be the culprit. To troubleshoot, I recommend using your browser's developer tools, which will show you the source of all the styles applied to the element (as you saw in **Chapter 11**). Or, if you don't want to have to worry about browser styles at all, one solution is to reset the padding and margins for all elements to zero. This is discussed in the "CSS Reset" section in **Chapter 21, More CSS Techniques**.

element. If that element's width changes, so will the margins on all four sides of the child element (padding has this behavior as well).

The **auto** keyword allows the browser to apply the amount of margin necessary to fit or fill the available space (see **Tip**). Adding a margin to the **body** element adds space between the page content and the edges of the viewport.

FIGURE 15-16 shows the results of the following margin examples. I've added a red border to the elements in the examples to make their boundaries clearer. The dotted rules were added in the figure illustration to indicate the outer edges of the margins for clarity purposes only; they are not something you'd see in the browser.

A
```
p#A {
   margin: 4em;
   border: 2px solid red;
   background: #e2f3f5;
}
```

B
```
p#B {
   margin-top: 2em;
   margin-right: 250px;
   margin-bottom: 1em;
   margin-left: 4em;
   border: 2px solid red;
   background: #e2f3f5;
}
```

C
```
body {
   margin: 0 20%;
   border: 3px solid red;
   background-color: #e2f3f5;
}
```

Adding a margin to the body element adds space between the page content and the edges of the viewport.

Take a look at example **C** in FIGURE 15-16. Here, I've applied the **margin** property to the **body** element of the document. For this particular design, I set the top margin to zero (0) so the **body** starts flush against the top edge of the browser window. Adding equal amounts of margin to the left and right sides of the **body** keeps the content of the page centered and gives it a little breathing room.

■ TIP

Centering with auto Margins

Setting the margin to **auto** on the left and right sides of a sized element has the effect of centering the element in its container.

Margin Behavior

Writing rules that apply margins around HTML elements is fairly straightforward, but it is important to be familiar with some of the quirks of margin behavior.

Collapsing margins

The most significant margin behavior to be aware of is that the top and bottom margins of neighboring elements **collapse**. This means that instead of accumulating, adjacent margins overlap, and only the largest value is used.

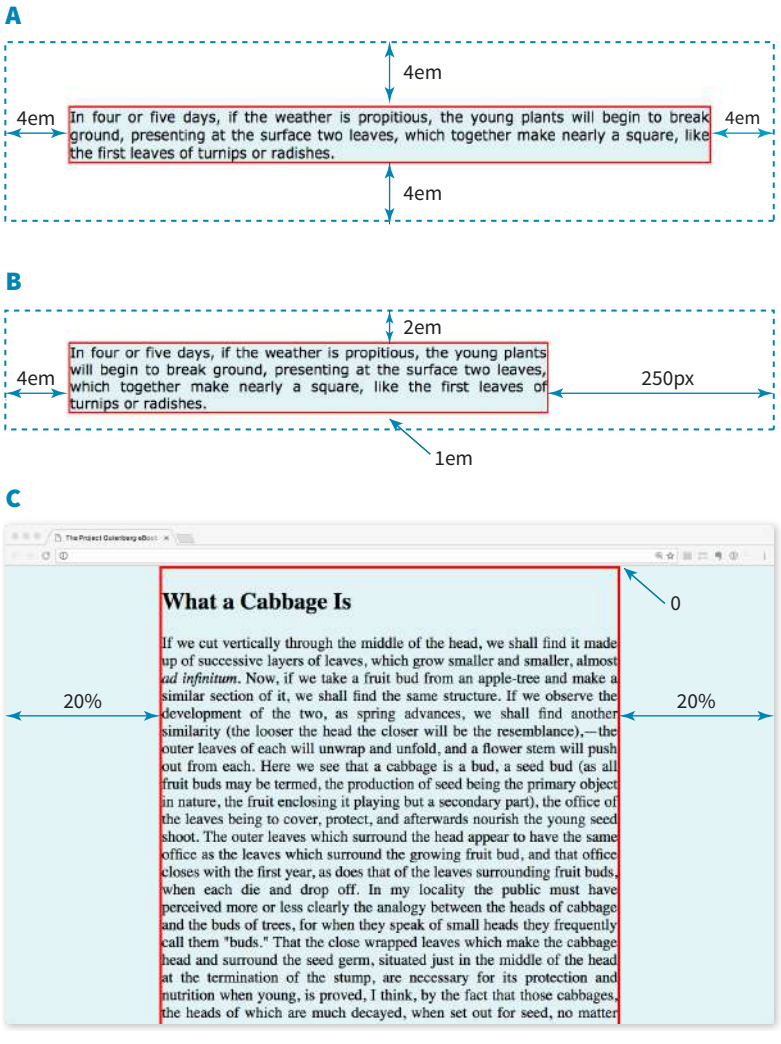

A
4em
4em
In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.
4em
4em
B
2em
4em
In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.
250px
1em
C
What a Cabbage Is
If we cut vertically through the middle of the head, we shall find it made up of successive layers of leaves, which grow smaller and smaller, almost ad infinitum. Now, if we take a fruit bud from an apple-tree and make a similar section of it, we shall find the same structure. If we observe the development of the two, as spring advances, we shall find another similarity (the looser the head the closer will be the resemblance),—the outer leaves of each will unwrap and unfold, and a flower stem will push out from each. Here we see that a cabbage is a bud, a seed bud (as all fruit buds may be termed, the production of seed being the primary object in nature, the fruit enclosing it playing but a secondary part), the office of the leaves being to cover, protect, and afterwards nourish the young seed shoot. The outer leaves which surround the head appear to have the same office as the leaves which surround the growing fruit bud, and that office closes with the first year, as does that of the leaves surrounding fruit buds, when each die and drop off. In my locality the public must have perceived more or less clearly the analogy between the heads of cabbage and the buds of trees, for when they speak of small heads they frequently call them "buds." That the close wrapped leaves which make the cabbage head and surround the seed germ, situated just in the middle of the head at the termination of the stump, are necessary for its protection and nutrition when young, is proved, I think, by the fact that those cabbages, the heads of which are much decayed, when set out for seed, no matter
0
20%
20%

FIGURE 15-16. Applying margins to the body and to individual elements

Using the paragraphs in FIGURE 15-16 **A** and **B** as an example, if the top element has a bottom margin of 4em and the following element has a top margin of 2em, the resulting margin space between elements does not add up to 6em. Rather, the margins collapse, so the resulting margin between the paragraphs will be 4em (the largest specified value). This is demonstrated in FIGURE 15-17.

Adjacent margins overlap, and only the largest value is used.

The only time top and bottom margins *don't* collapse is with floated or absolutely positioned elements (we'll get to those in **Chapter 16, Floating and Positioning**) and flex items (discussed in **Chapter 17**). Margins on the left and right sides never collapse, so they're nice and predictable.

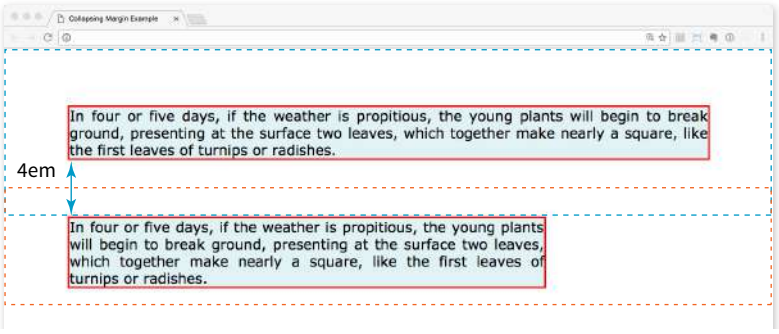

In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.
4em
In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.

FIGURE 15-17. Vertical margins of neighboring elements collapse, so that only the larger value is used.

When spacing between and around elements behaves unpredictably, collapsing margins are often to blame. For that reason, many developers pick one side (`margin-top` or `margin-bottom`) to apply margins to for most elements on the page, instead of applying them to both. To learn more about this aspect of margin behavior, I recommend reading the following articles (they were written long, long ago, but the principles still apply):

- "Uncollapsing Margins" by Eric Meyer (*oreil.ly/61NTE*)
- "What You Should Know about Collapsing Margins" by Geoff Graham (*oreil.ly/F5VeG*)

Margins on inline elements

You can apply top and bottom margins to inline text elements (or "non-replaced inline elements," to use the proper CSS terminology), but it won't have any visible effect. The browser won't add vertical space above and below the element, and the height of the line will not change. However, when you apply left and right margins to inline text elements, margin space *will* be held clear before and after the text in the flow of the element, even if that element breaks over several lines.

Margins on replaced inline elements, such as images, *do* render on all sides, and therefore do affect the height of the line. See FIGURE 15-18 for examples of each.

Negative margins

It is worth noting that it is possible to specify negative values for margins. When you apply a negative margin, the content, padding, and border are moved in the opposite direction that would have resulted from a positive margin value.

I'll make this clear with an example. FIGURE 15-19 shows two neighboring paragraphs with different-colored borders applied to show their boundaries.

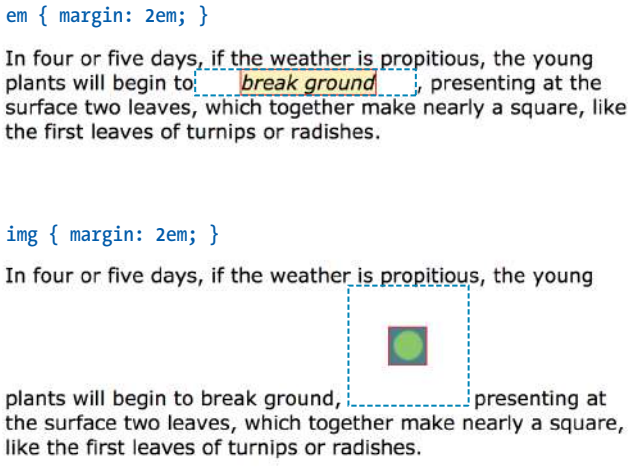

em { margin: 2em; }
In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.
img { margin: 2em; }
In four or five days, if the weather is propitious, the young
plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.

FIGURE 15-18. Examples of margins on inline elements. Only horizontal margins are rendered on non-replaced elements (top). Margins are rendered on all sides of replaced elements, such as images.

In the positive value view, I've added a 3em bottom margin to the top paragraph, which has the effect of pushing the following paragraph *down* by that amount. If I specify a negative value (–3em), the following element moves *up* by that amount and overlaps the element with the negative margin.

This may seem like a strange thing to do, and in fact, you probably wouldn't make blocks of text overlap as shown in this figure. The point here is that you can use margins with both positive and negative values to move elements around on the page.

Now, let's use margins to add some space between parts of the Black Goose Bakery home page in EXERCISE 15-3.

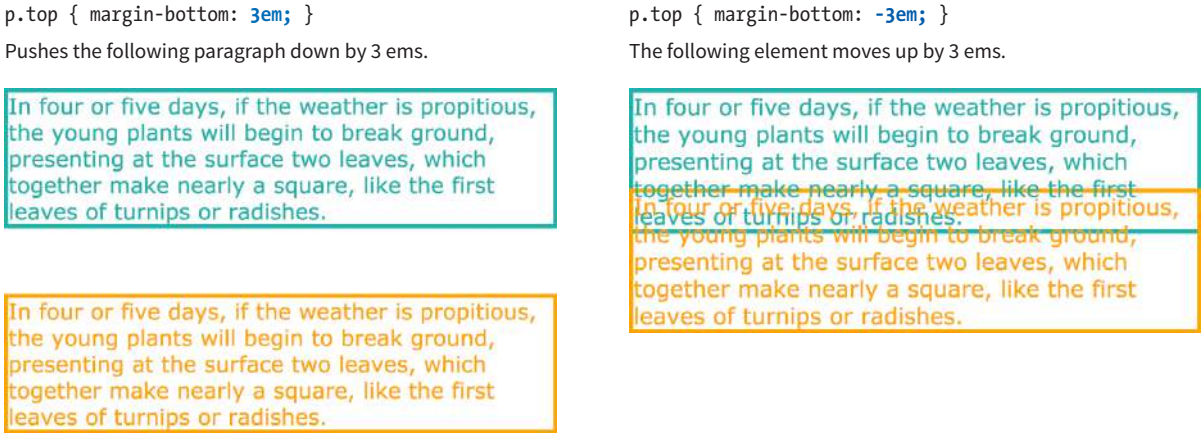

p.top { margin-bottom: 3em; }
Pushes the following paragraph down by 3 ems.
In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.
In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.
p.top { margin-bottom: -3em; }
The following element moves up by 3 ems.
In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.
In four or five days, if the weather is propitious, the young plants will begin to break ground, presenting at the surface two leaves, which together make nearly a square, like the first leaves of turnips or radishes.

FIGURE 15-19. Using negative margins

EXERCISE 15-3. Adding margin space around elements

It's time to adjust the margins around the elements on the bakery page. We'll start by adjusting the margins for the whole document, and then make tweaks to each section from top to bottom. You should have *bakery-styles.css* open in a text editor.

1. It's a common practice to set the margin for the **body** element to zero, thus clearing out the browser's default margin setting. Add this margin declaration to the **body** styles, and then save the file and open *bakery.html* in a browser—you'll see that the elements now extend to the very edge of the window with no space in between:

   ```
   body {
     ...
     margin: 0;
   }
   ```

   NOTE: When the value is 0, you don't need to provide a speciric unit.

2. If you're a careful observer, you may have noticed that there is still a bit of whitespace above the colored navigation bar. That happens to be the top margin of the `ul` list pushing the whole **nav** element down from the top edge of the browser window. Let's take care of that. Add a new style rule in the "nav styles" section of the style sheet:

   ```
   nav ul {
     margin: 0;
   }
   ```

3. Margins are good for nudging elements around in the layout. For example, I'd like to move the `h1` with the logotype down a bit by adding a margin to its top edge. I played around with a few values before deciding on 36px for this new style rule (you can try out some different values too):

   ```
   header h1 {
     margin-top: 36px;
   }
   ```

   I'd also like the intro paragraph in the header to be a little closer to the logotype, so let's get wacky and use a *negative* top margin to pull it *up*. Add this declaration to the existing style rule:

   ```
   header p {
     ...
     margin-top: -12px;
   }
   ```

4. Next, give the `main` section a 2.5% margin on all sides:

   ```
   main {
     ...
     margin: 2.5%;
   }
   ```

   Save the file and reload the page in the browser. Then try resizing the window, paying attention to how the space all around the `main` section (even the top edge) grows when the viewport gets wider. If you don't enjoy that effect, you can reset the margin to a stable px value instead, but I wanted you to get a feel for how a percentage value works. Let's also add a little extra space above the `h3` headings in the main area. I've chosen 2.5em, but you can play around with different values to see what you like best:

   ```
   h3 {
     ...
     margin-top: 2.5em;
   }
   ```

5. Finally, let's add some margins around the `aside`. We'll use different amounts and units on each side, for kicks. Put 1em on the top, 2.5% on the right side, 0 on the bottom, and a 75px margin on the left. I'm going to let you do this one yourself. Can you make all those changes with one declaration? If you want to check your work, the finished version of the Black Goose Bakery page so far is available with the exercise materials for this chapter.

Save the style sheet again, and reload the page. It should look like FIGURE 15-20. This isn't the most beautiful design, particularly if your browser window is set wide. However, if you make your browser window narrower, you'll find that it wouldn't be too bad as the small-screen version in a responsive design.

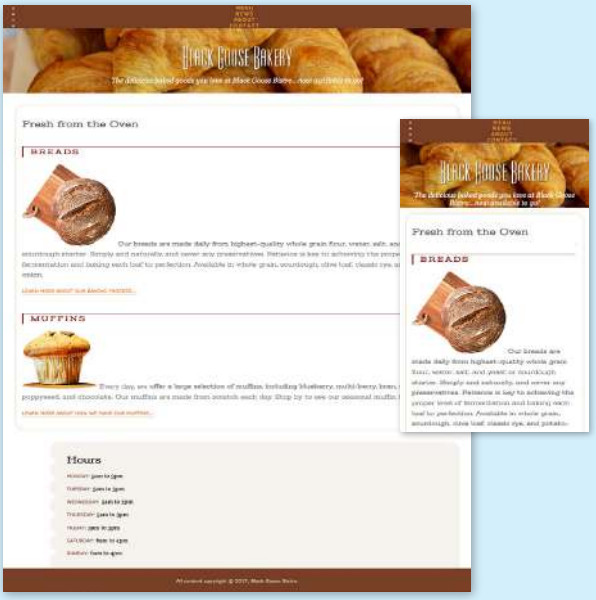


FIGURE 15-20. The Black Goose Bakery home page after padding, borders, and margins are added

ASSIGNING DISPLAY TYPES

As long as we're talking about boxes and the CSS layout model, this is a good time to introduce the **display** property. You should already be familiar with the display behavior of block and inline elements. Although HTML assigns display behaviors (or **display types**, to use the latest CSS term) to the elements it defines, there are other XML-based languages that use CSS that don't do the same. For this reason, the **display** property was created to allow authors to specify how elements should behave in layouts.

display

Values: Single-keyword values:

Legacy single-keyword values: block | inline | flow-root | inline-block | flex | inline-flex | grid | inline-grid

Box generation: none | contents

Other single-keyword values: list-item | inline-list-item | table | inline-table| table-row | table-cell | table-row-group | table-header-group | table-footer-group | table-column | table-column-group | table-caption | run-in | ruby-text (and related ruby display types)

Multi-keyword values:

Outer keywords: block | inline

Inner keywords: flow | flow-root | flex | grid | list-item | table | ruby

Default: inline

Applies to: All elements

Inherits: Yes

The **display** property defines the type of element box an element generates in the layout. In addition to the familiar **inline** and **block** display types, you can also make elements display as flexboxes, grids, list items, or the various parts of a table (you'll get some practice with the first two when we discuss using flexboxes and grids for page layout in **Chapters 17** and **18**). There are also a number of values for ruby annotation for East Asian languages. As you can see from the list of values, there are a lot of display types, but there are only a few that you'll use in everyday practice.

Display type assignment is useful for achieving layout effects while keeping the semantics of the HTML source intact. For example, it is common practice to make **li** elements (which usually display with the characteristics of block elements) display as inline elements to turn a list into a horizontal navigation bar, and you can make an otherwise inline **a** (anchor) element display as a block in order to give it a specific width and height:

```
ul.navigation li { display: inline; }
ul.navigation li a { display: block; }
```

The newer **flow-root** display type establishes the element as a block container with normal flow content (see **Note**).

WARNING

Bear in mind that changing the presentation of an HTML element with the CSS **display** *property does not change the definition of that element as block-level or phrasing content in HTML. Putting a block-level element within a phrase element will always be invalid, regardless of its display type.*

NOTE

In **Chapter 16, Floating and Positioning**, *you'll learn how the* **flow-root** *display type can be used to prevent elements from collapsing when all of their contents have been floated to the left or right, but I don't want to get too ahead of myself here.*

Another useful value for the `display` property is `none`, which removes the content from the normal flow entirely. Unlike `visibility: hidden`, which makes the element invisible but keeps the space it would have occupied blank, `display: none` removes the content, and the space it would have occupied is closed up.

One popular use of `display: none` is to prevent certain content in the source document from displaying in specific media, such as when the page is printed or displayed on devices with small screens. For example, you could display URLs for links in a document when it is printed, but not when it is displayed on a computer screen where the links are interactive.

Be aware that content that has its `display` property set to `none` still downloads with the document. Setting some content to `display: none` for devices with small screens may keep the page more compact, but it won't do anything to reduce data usage or download times.

The other issue with `display: none` is that it makes the hidden text inaccessible by screen readers. If you want to hide text but still have it read by screen readers, there is a hack that uses the `clip` property to reduce the element to one pixel on the page. The clip technique is described in the WebAIM article listed in the **"Further Reading"** sidebar.

> **FURTHER READING**
>
> "10 Ways to Hide Elements in CSS" by Baljeet Rathi (*sitepoint.com/hide-elements-in-css*) compares various methods for hiding content, including `display: none`.
>
> WebAIM's "Invisible Content Just for Screen Reader Users" (*webaim.org/techniques/css/invisiblecontent/*) lists the pros and cons of various methods for hiding text from an accessibility standpoint.

Multikeyword display Values

CSS Display Module Level 3 introduced a new multikeyword syntax for specifying display type. It is based on an awareness that elements have both an "outer" display behavior that describes how they appear in the surrounding flow (for example, block or inline) and an "inner" display behavior that describes the display of their contents (for example, a table, list item, grid, or flow root).

The new multiword syntax uses a combination of outer keywords (`inline` and `block`) and inner keywords (`flow`, `flow-root`, `flex`, `grid`, `list-item`, `table`, and `ruby`). So, where before we used `display: inline-table`, now we can use an outer and inner keyword combination, `display: inline table`. The legacy `display: block` is the equivalent of `display: block flow-root`. TABLE 15-1 lists multikeyword equivalents of the most useful single keywords.

As of this writing, the multikeyword syntax is not well supported, so it is recommended that you stick with the legacy keywords, which will continue working even as the new syntax becomes a more reliable option. It is worth knowing about, however, because it is a more flexible system for accommodating whatever new display types may emerge in the future. For more detailed information about the multikeyword display syntax, take a look at these resources:

- CSS Display Module Level 3 (*w3.org/TR/css-display-3/*)
- "Adapting to the New Multi-Keyword Syntax of display" (*oreil.ly/E3Wtv*)

TABLE 15-1. Single keywords and their multikeyword equivalents

Single keyword	Multikeyword equivalent
block	block flow-root
inline	inline flow
inline-block	inline flow-root
flow-root	block flow-root
flex	block flex
inline-flex	inline flex
grid	block grid
inline-grid	inline grid

BOX DROP SHADOWS

We've arrived at the last stop on the element box tour. In **Chapter 12, Formatting Text**, you learned about the **text-shadow** property, which adds a drop shadow to text. The **box-shadow** property applies a drop shadow around the entire visible element box (excluding the margin).

box-shadow

Values: *horizontal-offset vertical-offset blur-distance spread-distance color* inset | none

Default: none

Applies to: All elements

Inherits: No

The **box-shadow** property should seem familiar to you after working with **text-shadow**: you specify the horizontal and vertical offset distances, the amount the shadow should blur, and a color. For box shadows, you can additionally specify a **spread** amount, which increases (or decreases with negative values) the size of the shadow. By default, the shadow color is the same as the foreground color of the element, but specifying a color overrides that.

FIGURE 15-21 shows the results of the following code examples. **A** adds a simple box shadow 6 pixels to the right and 6 pixels down, without blur or spread. **B** adds a blur value of 5 pixels, and **C** shows the effect of a 10-pixel spread value:

```
A box-shadow: 6px 6px gray;
B box-shadow: 6px 6px 5px gray; /* 5-pixel blur */
C box-shadow: 6px 6px 5px 10px gray; /* 5px blur, 10px spread */
```

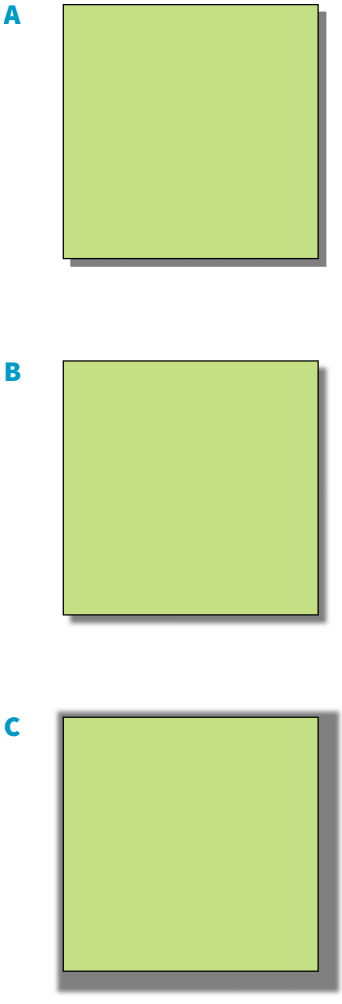


FIGURE 15-21. Adding drop shadows around an element with the **box-shadow** property

Box shadows are always applied to the area *outside* the border of the element by default (or the place it would be if a border isn't specified). If the element has a transparent or translucent background, no box shadow color will be visible in the content and padding area of the element.

You can make the shadow render inside the edges of the visible element box by adding the **inset** keyword to the rule. This makes it look like the element is pressed into the screen (FIGURE 15-22):

```
box-shadow: inset 6px 6px 5px gray;
```

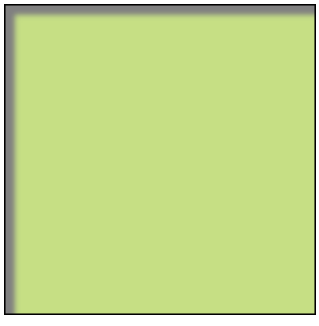


FIGURE 15-22. An inset box shadow renders on the inside of the element box.

As with **text-shadow**, you can specify multiple box shadows on an element by providing the values in a comma-separated list, which can produce some nuanced or funky effects. The values that come first get placed on top, and subsequent shadows are placed behind it in the order in which they appear in the list.

WARNING

Box shadows, text shadows, and gradients take a lot of processor power because you are shifting the burden of interpreting and rendering them onto the browser. The more you use, the slower performance will be, and as we all know, performance is everything on the web. So, go easy on these.

TEST YOURSELF

At this point, you should have a good feel for element boxes and how to manipulate the space within and around them. In the next chapter, we'll start moving the boxes around on the page—but first, why not get some practice at writing rules for padding, borders, and margins in the following test?

In this test, your task is to write the declarations that create the effects shown in each example in FIGURE 15-23 (see **Useful Hints**). All the paragraphs shown here share a rule that sets the dimensions and the background color for each paragraph. You just need to provide the box-related property declarations. Answers, as always, appear in **Appendix A**.

USEFUL HINTS

- Outer margin edges are indicated by dotted blue lines.
- All necessary measurements are provided in blue.
- Borders are either `black` or `red`.

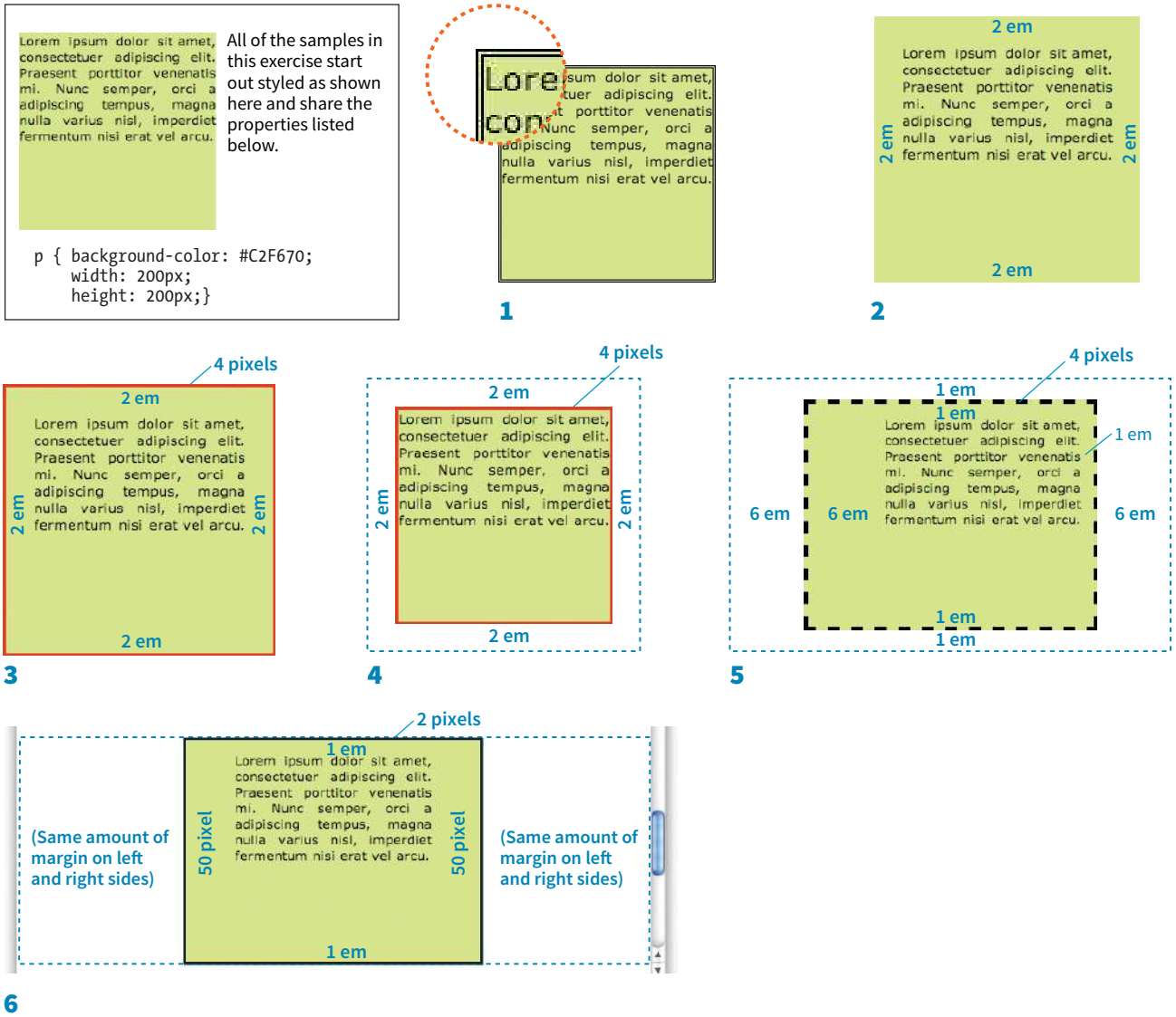


FIGURE 15-23. Write the declarations for these examples.

CSS REVIEW: BOX PROPERTIES

Here is a roundup of the many CSS properties related to box dimensions, borders, padding, margins, and outlines.

Property	Description
`block-size`	Specifies the size of the element perpendicular to the writing direction
`border`	Shorthand property that combines border properties

`border-top`
`border-right`
`border-bottom`
`border-left`
`border-block-start`
`border-block-end`
`border-inline-start`
`border-inline-end`
`border-block`
`border-inline`
	Shorthand properties that combine border properties for each side of the element

`border-color`	Shorthand property for specifying the color of borders

`border-top-color`
`border-right-color`
`border-bottom-color`
`border-left-color`
`border-block-start-color`
`border-block-end-color`
`border-inline-start-color`
`border-inline-end-color`
`border-block-color`
`border-inline-color`
	Specify the border color for each side of the element

`border-image`	Adds an image inside the border area
`border-image-outset`	Specifies how far the border image should be positioned away from the content area
`border-image-repeat`	Specifies the manner in which the image fills the border
`border-image-slice`	Specifies the point at which an image should be divided in order to be applied to the corners and sides of the border
`border-image-source`	Specifies the location of the image file to be used for the border image
`border-image-width`	Specifies the width of the space the border image should occupy
`border-radius`	Shorthand property for rounding the corners of the visible element box

`border-top-left-radius`
`border-top-right-radius`
`border-bottom-right-radius`
`border-bottom-left-radius`
`border-start-start-radius`
`border-start-end-radius`
`border-end-end-radius`
`border-end-start-radius`
	Specify the radius curve for each individual corner

Property	Description
`border-style`	Shorthand property for specifying the style of borders

`border-top-style`
`border-right-style`
`border-bottom-style`
`border-left-style`
`border-block-start-style`
`border-block-end-style`
`border-inline-start-style`
`border-inline-end-style`
`border-block-style`
`border-inline-style`
	Specify the border style for each side of the element

`border-width`	Shorthand property for specifying the width of borders

`border-top-width`
`border-right-width`
`border-bottom-width`
`border-left-width`
`border-block-start-width`
`border-block-end-width`
`border-inline-start-width`
`border-inline-end-width`
`border-block-width`
`border-inline-width`
	Specify the border width for each side of the element

`box-sizing`	Specifies whether width and height dimensions apply to the content box or the border box
`box-shadow`	Adds a drop shadow around the visible element box
`display`	Defines the type of element box an element generates
`height`	Specifies the height of the element's content box or border box
`inline-size`	Specifies a length in the direction of the writing flow
`margin`	Shorthand property for specifying margin space around an element

`margin-top`
`margin-right`
`margin-bottom`
`margin-left`
`margin-block-start`
`margin-block-end`
`margin-inline-start`
`margin-inline-end`
`margin-block`
`margin-inline`
	Specify the margin amount for each side of the element

`max-height`	Specifies the maximum height of an element
`max-width`	Specifies the maximum width of an element
`min-height`	Specifies the minimum height of an element
`min-width`	Specifies the minimum width of an element

CSS Box Properties, *Continued*

Property	Description
`outline`	Shorthand property for adding an outline around an element
`outline-color`	Sets the color of the outline
`outline-offset`	Sets the space between the outline and the outer edge of the border
`outline-style`	Sets the style of the outline
`outline-width`	Sets the width of the outline
`overflow`	Specifies how to handle content that doesn't fit in the content area
`padding`	Shorthand property for specifying space between the content area and the border

`padding-top`
`padding-right`
`padding-bottom`
`padding-left`
`padding-block-start`
`padding-block-end`
`padding-inline-start`
`padding-inline-end`
`padding-block`
`padding-inline`
	Specify the padding amount for each side of the element

`width`	Specifies the width of an element's content box or border box

CHAPTER
16

FLOATING AND POSITIONING

IN THIS CHAPTER

Floating elements to the left and right

Clearing floated elements

Containing floated elements

Creating text-wrap shapes

Relative positioning

Absolute positioning and containing blocks

Fixed positioning

At this point, you've learned about dozens of CSS properties that let you change the appearance of text elements and the boxes they generate. But so far, we've merely been formatting elements as they appear in the normal flow of the document.

In this chapter, we'll look at floating and positioning, the CSS methods for breaking out of the normal flow and arranging elements on the page. **Floating** an element moves it to the left or right and allows the following text to wrap around it. **Positioning** is a way to specify the location of an element anywhere on the page with pixel precision. These features are used less often now that we have Flexbox and Grid Layout, but they are still useful tools worth knowing about.

Before we start moving elements around, though, let's be sure we're well acquainted with how they behave in the normal flow.

NORMAL FLOW

We've covered the normal flow in previous chapters, but it's worth a refresher (see **Note**). In the CSS layout model, text elements are laid out from top to bottom in the order in which they appear in the source, and from left to right in left-to-right reading languages (for right-to-left reading languages such as Arabic and Hebrew, the normal flow is top to bottom and right to left).

Block elements stack up on top of one another and fill the available width of the viewport or other containing element. Inline elements and text characters line up next to one another to fill the block elements. When the window or containing element resizes, the block elements expand or contract to the new width and the inline content reflows to fit, as shown in FIGURE 16-1.

NOTE

In this chapter, I'll be referencing the normal flow as it behaves for languages written horizontally, such as English. Keep in mind that the normal flow works differently for vertically written languages.

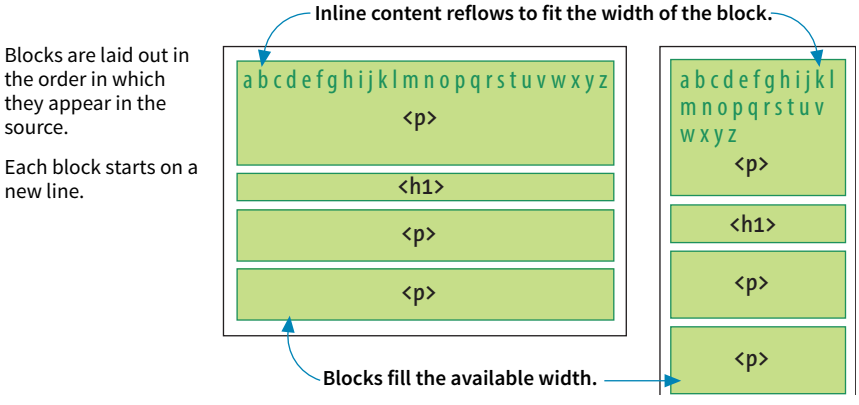

Inline content reflows to fit the width of the block.
Blocks are laid out in the order in which they appear in the source.
Each block starts on a new line.
a b c d e f g h i j k l m n o p q r s t u v w x y z
<p>
<h1>
<p>
<p>
Blocks fill the available width.
a b c d e f g h i j k l m n o p q r s t u v w x y z
<p>
<h1>
<p>
<p>

FIGURE 16-1. One more example of the normal flow behavior

Objects in the normal flow affect the layout of the objects around them. This is the behavior we've come to expect in web pages—elements don't overlap or bunch up; they make room for one another.

We've seen all of this before, but in this chapter we'll be paying attention to whether elements are in the flow or removed from the flow. Floating and positioning change the relationship of elements to the normal flow in different ways. Let's first look at the special behavior of floated elements (or "floats" for short).

FLOATING

Floating an element moves it to the left or right and allows the following text to wrap around it.

Simply stated, the **`float`** property moves an element as far as possible to the left or right (to the beginning or end of the line, to put it logically), allowing the following content to wrap around it. It is a unique feature built into CSS with some interesting behaviors.

`float`

Values: `left` | `right` | `none` | `inline-start` | `inline-end`

Default: none

Applies to: All elements

Inherits: No

The best way to explain floating is to demonstrate it. In this example, the **`float`** property is applied to an **`img`** element to float it to the right. FIGURE 16-2 shows how the paragraph and the contained image are rendered by default (top) and how it looks when the **`float`** property is applied (bottom).

WARNING

The logical values `inline-start` *and* `inline-end` *(equivalent to* `left` *and* `right` *in languages written left to right) have only been supported by browsers since 2024, so it may be a little while before they can be used reliably without fallbacks.*

THE MARKUP

```
<p><img src="icecreambowl.png" alt=""> After the cream is frozen rather
stiff, ...
```

THE STYLES

```
img {
  float: right;
}
```

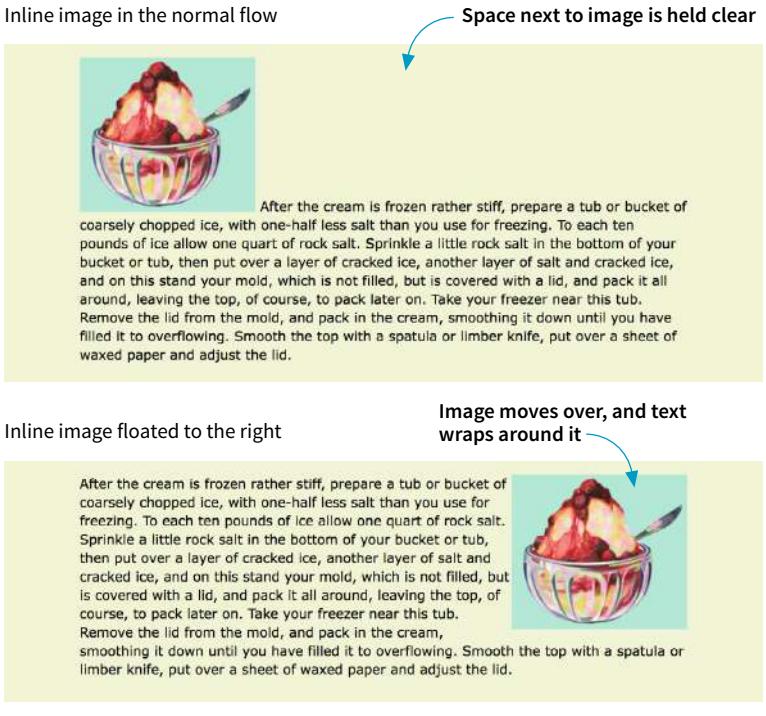

Inline image in the normal flow
Space next to image is held clear
After the cream is frozen rather stiff, prepare a tub or bucket of coarsely chopped ice, with one-half less salt than you use for freezing. To each ten pounds of ice allow one quart of rock salt. Sprinkle a little rock salt in the bottom of your bucket or tub, then put over a layer of cracked ice, another layer of salt and cracked ice, and on this stand your mold, which is not filled, but is covered with a lid, and pack it all around, leaving the top, of course, to pack later on. Take your freezer near this tub. Remove the lid from the mold, and pack in the cream, smoothing it down until you have filled it to overflowing. Smooth the top with a spatula or limber knife, put over a sheet of waxed paper and adjust the lid.
Inline image floated to the right
Image moves over, and text wraps around it
After the cream is frozen rather stiff, prepare a tub or bucket of coarsely chopped ice, with one-half less salt than you use for freezing. To each ten pounds of ice allow one quart of rock salt. Sprinkle a little rock salt in the bottom of your bucket or tub, then put over a layer of cracked ice, another layer of salt and cracked ice, and on this stand your mold, which is not filled, but is covered with a lid, and pack it all around, leaving the top, of course, to pack later on. Take your freezer near this tub. Remove the lid from the mold, and pack in the cream, smoothing it down until you have filled it to overflowing. Smooth the top with a spatula or limber knife, put over a sheet of waxed paper and adjust the lid.

FIGURE 16-2. The layout of an image in the normal flow (top) and with the `float` property applied (bottom)

That's a nice effect; I've gotten rid of a lot of wasted space on the page. But now the text is bumping right up against the image. How do you think you would add some space between the image element and the surrounding text? If you guessed "add a margin," you're absolutely right. I'll add 1em of space on all sides of the image with the **`margin`** property (FIGURE 16-3). You can begin to see how the box properties work together to improve page layout:

```
img {
  float: right;
  margin: 1em;
}
```

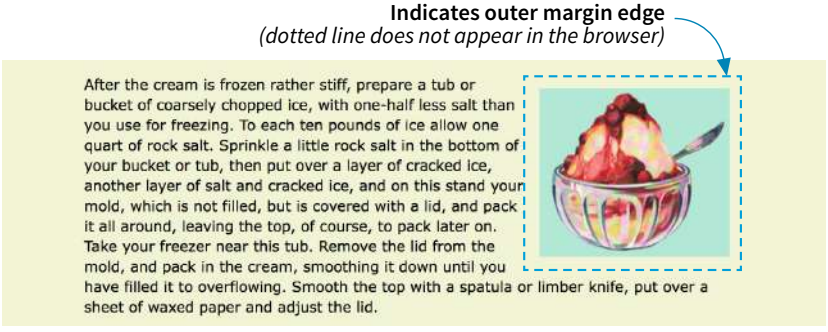

Indicates outer margin edge
(dotted line does not appear in the browser)
After the cream is frozen rather stiff, prepare a tub or bucket of coarsely chopped ice, with one-half less salt than you use for freezing. To each ten pounds of ice allow one quart of rock salt. Sprinkle a little rock salt in the bottom of your bucket or tub, then put over a layer of cracked ice, another layer of salt and cracked ice, and on this stand your mold, which is not filled, but is covered with a lid, and pack it all around, leaving the top, of course, to pack later on. Take your freezer near this tub. Remove the lid from the mold, and pack in the cream, smoothing it down until you have filled it to overflowing. Smooth the top with a spatula or limber knife, put over a sheet of waxed paper and adjust the lid.

FIGURE 16-3. Adding a 1em margin around the floated image

The previous two figures demonstrate key behaviors of floated elements:

A floated element is like an island in a stream.

First and foremost, you can see that the image is removed from its position in the normal flow yet continues to influence the surrounding content. The subsequent paragraph text reflows to make room for the floated `img` element. One popular analogy compares floats to islands in a stream—they are not in the flow, but the stream has to flow around them. This behavior is unique to floated elements.

Floats stay in the content area of the containing element.

It is also important to note that the floated image is placed within the *content area* (the inner edges) of the paragraph that contains it. It does not extend into the padding area of the paragraph.

Margins are maintained.

In addition, margins are held on all sides of the floated image, as indicated in FIGURE 16-3 by the dotted line. In other words, the entire element box, from outer edge to outer edge, is floated.

Floating Inline and Block Elements

Those are the basics, so now let's look at more examples and explore additional floating behaviors. It is possible to float any HTML element, both inline and block-level, as you'll see in the following examples.

Floating an inline text element

In the previous example, we floated an inline image element. This time, let's look at what happens when we float an inline text (non-replaced) element—in this case, a span of text (FIGURE 16-4).

THE MARKUP

```
<p><span class="tip">TIP: Make sure that your packing tub or bucket
has a hole below the top of the mold so the water will drain
off.</span>After the cream is frozen rather stiff, prepare a tub...</p>
```

THE STYLES

```
.tip {
  float: right;
  margin: 1em;
  width: 200px;
  color: #fff;
  background-color: lightseagreen;
  padding: 1em;
}
```

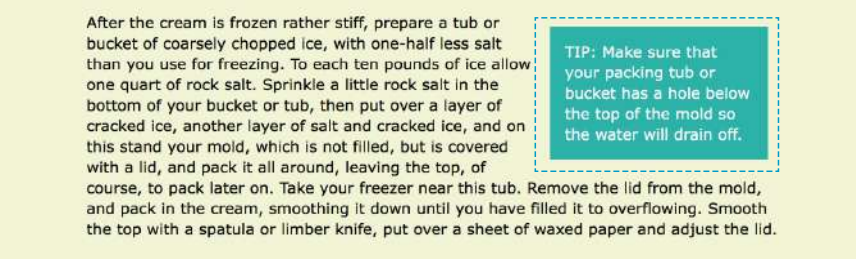

After the cream is frozen rather stiff, prepare a tub or bucket of coarsely chopped ice, with one-half less salt than you use for freezing. To each ten pounds of ice allow one quart of rock salt. Sprinkle a little rock salt in the bottom of your bucket or tub, then put over a layer of cracked ice, another layer of salt and cracked ice, and on this stand your mold, which is not filled, but is covered with a lid, and pack it all around, leaving the top, of course, to pack later on. Take your freezer near this tub. Remove the lid from the mold, and pack in the cream, smoothing it down until you have filled it to overflowing. Smooth the top with a spatula or limber knife, put over a sheet of waxed paper and adjust the lid.
TIP: Make sure that your packing tub or bucket has a hole below the top of the mold so the water will drain off.

FIGURE 16-4. Floating an inline text (non-replaced) element. The dotted lines indicate the outer edge of the element and would not be visible in the browser.

At a glance, it is behaving the same as the floated image, which is what we'd expect. But there are some subtle things at work here that bear pointing out:

Always provide a width for floated text elements.

It is necessary to specify a width for floated text elements.

First, you'll notice that the style rule that floats the `span` includes the `width` property. It is necessary to specify a width for a floated text element because without one, its box is sized wide enough to fit its content (`auto`). For short phrases that are narrower than the container, that might not be an issue. However, for longer, wrapped text, the box expands to the width of the container, making it so wide that there won't be room to wrap anything around it. Images have an inherent width, so I didn't need to specify a width in the previous example (although I certainly could have).

Floated inline elements behave as block elements.

Notice that the margin is held on all four sides of the floated `span` text, even though top and bottom margins are usually not rendered on inline elements (see FIGURE 14-20 in the previous chapter). That is because all floated elements behave like block elements. Once you float an inline element, it follows the display rules for block-level elements, and margins are rendered on all four sides.

Margins on floated elements do not collapse.

In the normal flow, abutting top and bottom margins collapse (overlap), but margins for floated elements are maintained on all sides as specified.

Floating block elements

Now let's look at what happens when we float a block within the normal flow. In this example, the whole second paragraph element is floated to the left (FIGURE 16-5).

THE MARKUP

```
<p>If you wish to pack ice cream...</p>
<p class="floatexample">After the cream is frozen rather stiff...</p>
<p>Remove the lid from the mold...</p>
<p>Make sure that your packing tub or bucket...</p>
<p>As cold water is warmer than the ordinary...</p>
```

THE STYLES

```
p {
  border: 2px red solid;
}

.floatexample {
  float: left;
  width: 300px;
  margin: 1em;
  background: white;
}
```

If you wish to pack ice cream and serve it in forms or shapes, it must be molded after the freezing. The handiest of all of these molds is either the brick or the melon mold.
After the cream is frozen rather stiff, prepare a tub or bucket of coarsely chopped ice, with one-half less salt than you use for freezing. To each ten pounds of ice allow one quart of rock salt. Sprinkle a little rock salt in the bottom of your bucket or tub, then put over a layer of cracked ice, another layer of salt and cracked ice, and on this stand your mold, which is not filled, but is covered with a lid, and pack it all around, leaving the top, of course, to pack later on. Take your freezer near this tub.
Remove the lid from the mold, and pack in the cream, smoothing it down until you have filled it to overflowing. Smooth the top with a spatula or limber knife, put over a sheet of waxed paper and adjust the lid. Have a strip of muslin or cheese cloth dipped in hot paraffin or suet and quickly bind the seam of the lid. This will remove all danger of salt water entering the pudding. Now cover the mold thoroughly with ice and salt.
Make sure that your packing tub or bucket has a hole below the top of the mold, so that the salt water will be drained off. If you are packing in small molds, each mold, as fast as it is closed, should be wrapped in wax paper and put down into the salt and ice. These must be filled quickly and packed.
As cold water is warmer than the ordinary freezing mixture, after you lift the can or mold, wipe off the salt, hold it for a minute under the cold water spigot, then quickly wipe the top and bottom and remove the lid. Loosen the pudding with a limber knife, hold the mold a little slanting, give it a shake, and nine times out of ten it will come out quickly, having the perfect shape of the can or mold. If the cream still sticks and refuses to come out, wipe the mold with a towel wrung from warm water. Hot water spoils the gloss of puddings, and unless you know exactly how to use it, the cream is too much melted to garnish.
If you wish to pack ice cream and serve it in forms or shapes, it must be molded after the freezing. The handiest of all of these molds is either the brick or the melon mold.
After the cream is frozen rather stiff, prepare a tub or bucket of coarsely chopped ice, with one-half less salt than you use for freezing. To each ten pounds of ice allow one quart of rock salt. Sprinkle a little rock salt in the bottom of your bucket or tub, then put over a layer of cracked ice, another layer of salt and cracked ice, and on this stand your mold, which is not filled, but is covered with a lid, and pack it all around, leaving the top, of course, to pack later on. Take your freezer near this tub.
Remove the lid from the mold, and pack in the cream, smoothing it down until you have filled it to overflowing. Smooth the top with a spatula or limber knife, put over a sheet of waxed paper and adjust the lid. Have a strip of muslin or cheese cloth dipped in hot paraffin or suet and quickly bind the seam of the lid. This will remove all danger of salt water entering the pudding. Now cover the mold thoroughly with ice and salt.
Make sure that your packing tub or bucket has a hole below the top of the mold, so that the salt water will be drained off. If you are packing in small molds, each mold, as fast as it is closed, should be wrapped in wax paper and put down into the salt and ice. These must be filled quickly and packed.
As cold water is warmer than the ordinary freezing mixture, after you lift the can or mold, wipe off the salt, hold it for a minute under the cold water spigot, then quickly wipe the top and bottom and remove the lid. Loosen the pudding with a limber knife, hold the mold a little slanting, give it a shake, and nine times out of ten it will come out quickly, having the perfect shape of the can or mold. If the cream still sticks and refuses to come out, wipe the mold with a towel wrung from warm water. Hot water spoils the gloss of puddings, and unless you know exactly how to use it, the cream is too much melted to garnish.

FIGURE 16-5. Floating a block-level element

I've added a red border around the `p` elements to reveal their boundaries. In addition, I've made the background of the floated paragraph white so it stands out and added a 1em margin on all sides (indicated with a blue dotted line). The bottom view in FIGURE 16-5 shows how it looks with all the rules turned off, as it would more likely appear on a real page.

Just as we saw with the image, the paragraph moves off to the side (left this time) and the following content wraps around it, even though blocks normally stack on top of one another. There are a few things I want to point out in this example:

You must provide a width for floated block elements.

If you do not provide a `width` value, the width of the floated block will be set to `auto`, which fills the available width of the browser window or other containing element. There's not much sense in having a full-width floated box, because the idea is to wrap text next to the float, not start below it.

Elements do not float higher than their reference in the source.

A floated block will float to the left or right relative to where it occurs in the source, allowing the following elements in the flow to wrap around it. It stays below any block elements that precede it in the flow (in effect, it is "blocked" by them). That means you can't float an element up to the top corner of a page, even if its nearest ancestor is the `body` element. If you want a floated element to appear at the top of the page, it must appear first in the document source (see **Note**).

Nonfloated elements maintain the normal flow.

The red borders in the top image reveal that the element boxes for the surrounding paragraphs still extend the full width of the normal flow. Only the *content* of those elements wraps around the float. This is a good model to keep in mind, and it has some important consequences. For example, adding a left margin to the surrounding paragraphs would add space on the left edge of the page, not between the text and the floated element. If you want space between the float and the wrapped text, you need to apply the margin to the float itself.

NOTE

Absolute positioning is the CSS method for placing elements on a page regardless of where they appear in the source. We'll get to absolute positioning later in this chapter. You can also change the order in which elements display with the `order` *property in a flexbox or grid, as discussed in* **Chapter 17, Getting in Line with Flexbox**, *and* **Chapter 18, Grid Layout**.

Clearing Floated Elements

If you're going to be floating elements around, it's important to know how to turn the text wrapping *off* and get back to the normal flow. You do this by **clearing** the element that you want to start below the float. Applying the `clear` property to an element prevents it from appearing next to a floated element and forces it to start at the next available "clear" space below the float.

clear

Values: left | right | both | none | inline-start | inline-end

Default: none

Applies to: Block-level elements only

Inherits: No

Keep in mind that you apply the **clear** property to the element you want to start below the floated element, not to the floated element itself. The **left** value starts the element below any elements that have been floated to the left. Similarly, the **right** value makes the element clear all floats on the right edge of the containing block. If there are multiple floated elements and you want to be sure an element starts below all of them, use the **both** value to clear floats on both sides.

In this example, the **clear** property has been used to make **h2** elements start below left-floated elements. FIGURE 16-6 shows how the **h2** heading starts at the next available clear edge below the float:

```
img {
  float: left;
  margin-right: .5em;
}
h2 {
  clear: left;
  margin-top: 2em;
}
```

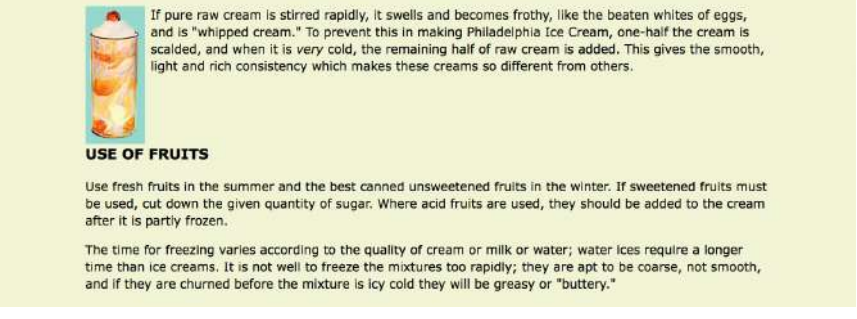

If pure raw cream is stirred rapidly, it swells and becomes frothy, like the beaten whites of eggs, and is "whipped cream." To prevent this in making Philadelphia Ice Cream, one-half the cream is scalded, and when it is very cold, the remaining half of raw cream is added. This gives the smooth, light and rich consistency which makes these creams so different from others.
USE OF FRUITS
Use fresh fruits in the summer and the best canned unsweetened fruits in the winter. If sweetened fruits must be used, cut down the given quantity of sugar. Where acid fruits are used, they should be added to the cream after it is partly frozen.
The time for freezing varies according to the quality of cream or milk or water; water ices require a longer time than ice creams. It is not well to freeze the mixtures too rapidly; they are apt to be coarse, not smooth, and if they are churned before the mixture is icy cold they will be greasy or "buttery."

FIGURE 16-6. Clearing a left-floated element

Notice in FIGURE 16-6 that although there is a 2em top margin applied to the **h2** element, it is not rendered between the heading and the floated image. That's the result of collapsing vertical margins in the flow. If you want to make sure space is held between a float and the following text, apply a bottom margin to the floated element itself.

By now you have enough float know-how to give it a try in EXERCISE 16-1.

EXERCISE 16-1. Floating images

In the exercises in this chapter, we'll make further improvements to the Black Goose Bakery home page that we worked on in **Chapter 15, Thinking Inside the Box**. If you did not follow along in the last chapter, or if you would just like a fresh start, there are copies of the documents in their most recent state with the materials for this chapter (*learningwebdesign.com/6e/materials*).

1. Open the CSS file *bakery-styles.css* in a text or code editor, and open *bakery.html* in a browser. We'll start by removing wasted vertical space next to the baked good images by floating those images to the left. Create a new style rule with a contextual selector to target only the images in the `main` section:

   ```
   main img {
     float: left;
   }
   ```

   Save the CSS file and refresh the page in the browser, and you'll see that we have some post-float tidying up to do.

2. We want the "Learn more" links to always appear below the images so they are clearly visible and consistently on the left side of the page. Fortunately, the paragraphs with those links are marked up with the class "more", and there is already a style rule for them using a class selector. Make those paragraphs clear any floats on the left edge:

   ```
   p.more {
     ...
     clear: left;
   }
   ```

3. Lastly, we'll adjust the spacing around the floated images. Give both images a 16px margin on the right and bottom sides by using the shorthand `margin` property:

   ```
   main img {
     float: left;
     margin: 0 16px 16px 0;
   }
   ```

   The muffin image could use extra space on the left side so it lines up better with the bread. Use this nifty attribute selector to grab any image whose `src` attribute contains the word *muffin* (there's only one):

   ```
   img[src*="muffin"] {
     margin-left: 50px;
   }
   ```

FIGURE 16-7 shows the new and improved "Fresh from the Oven" section.

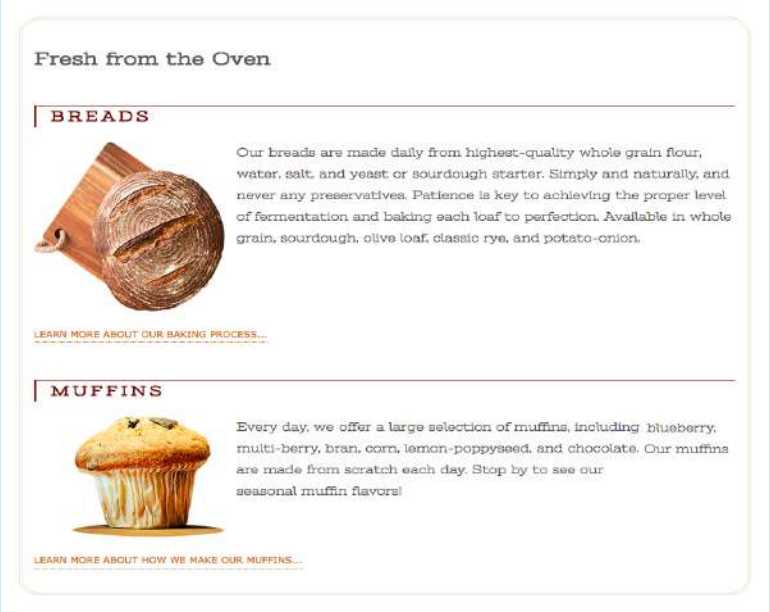

Fresh from the Oven
BREADS
Our breads are made daily from highest-quality whole grain flour, water, salt, and yeast or sourdough starter. Simply and naturally, and never any preservatives. Patience is key to achieving the proper level of fermentation and baking each loaf to perfection. Available in whole grain, sourdough, olive loaf, classic rye, and potato-onion.
LEARN MORE ABOUT OUR BAKING PROCESS...
MUFFINS
Every day, we offer a large selection of muffins, including blueberry, multi-berry, bran, corn, lemon-poppyseed, and chocolate. Our muffins are made from scratch each day. Stop by to see our seasonal muffin flavors!
LEARN MORE ABOUT HOW WE MAKE OUR MUFFINS...

FIGURE 16-7. The product section with floated images and wrapped text has less wasted space.

Floating Multiple Elements

It's perfectly fine to float multiple elements on a page, or even within a single element. In fact, before we had Flexbox and Grid Layout, floats were the primary method used for lining up elements like navigation menus and even for creating whole page layouts (thankfully, this approach is now obsolete).

Floated elements will be placed as far left or right and as high up as space allows.

When you float multiple elements, there is a complex system of behind-the-scenes rendering rules that ensures floated elements do not overlap. You can consult the CSS specification for details, but the upshot of it is that floated elements will be placed as far left or right (depending on the value you specify) and as high up as space allows.

FIGURE 16-8 shows what happens when a series of sequential paragraphs are floated to the same side. The first three floats start stacking up from the left edge, but when there isn't enough room for the fourth, it moves down and to the left until it bumps into something—in this case, the edge of the browser window. However, if one of the floats, such as P2, had been very long, it would have bumped up against the edge of the long float instead. Notice that the next paragraph in the normal flow (P6) starts wrapping at the highest point it can find, just below P1.

THE MARKUP

```
<p>[PARAGRAPH 1] ONCE upon a time...</p>
<p class="float">[P2]...</p>
<p class="float">[P3]...</p>
<p class="float">[P4]...</p>
<p class="float">[P5]...</p>
<p>[P6]...</p>
<p>[P7]...</p>
<p>[P8]...</p>
<p>[P9]...</p>
<p>[P10]...</p>
```

THE STYLES

```
p.float {
  float: left;
  width: 200px;
  margin: 0px;
  background: #F2F5d5;
  color: #DAEAB1;
}
```

Containing Floats

This is a good time to address a quirky float behavior: **float containment**. By default, floats are designed to hang out of the element they are contained in. That's just fine for allowing text to flow around a floated image, but sometimes it can cause some unwanted behaviors.

Take a look at the example in FIGURE 16-9. It would look nicer if the border contained all the content, but the floated image hangs out the bottom.

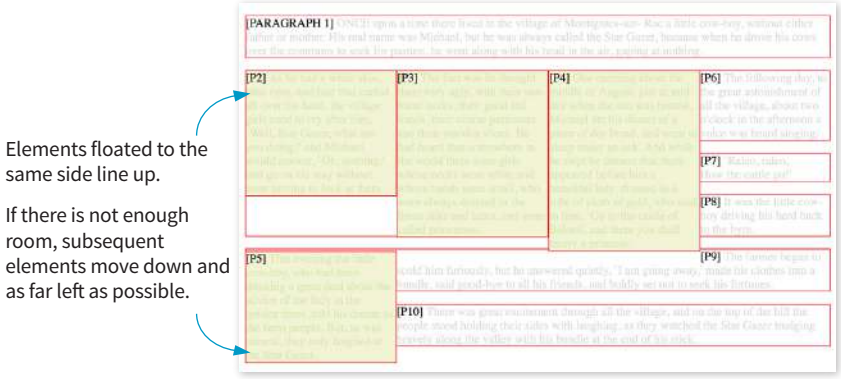

Elements floated to the same side line up.
If there is not enough room, subsequent elements move down and as far left as possible.

FIGURE 16-8. Multiple floated elements line up and do not overlap.

FIGURE 16-9. The containing element does not expand to accommodate the floated image, as indicated by its blue border.

If you float *all* the elements in a container element, there will be no elements remaining in the flow to hold the containing element open. This phenomenon is illustrated in FIGURE 16-10. The `#container div` contains two paragraphs. The view of the normal flow (top) shows that the `#container` has a background color and a border that wraps around the content.

THE MARKUP

```
<div id="container">
  <p>...</p>
  <p>...</p>
</div>
```

THE STYLES

```
#container {
  background: #f2f5d5;
  border: 2px dashed green;
}
```

Now suppose we add the following style rule:

```
p {
  float: left;
  width: 44%;
  padding: 2%;
}
```

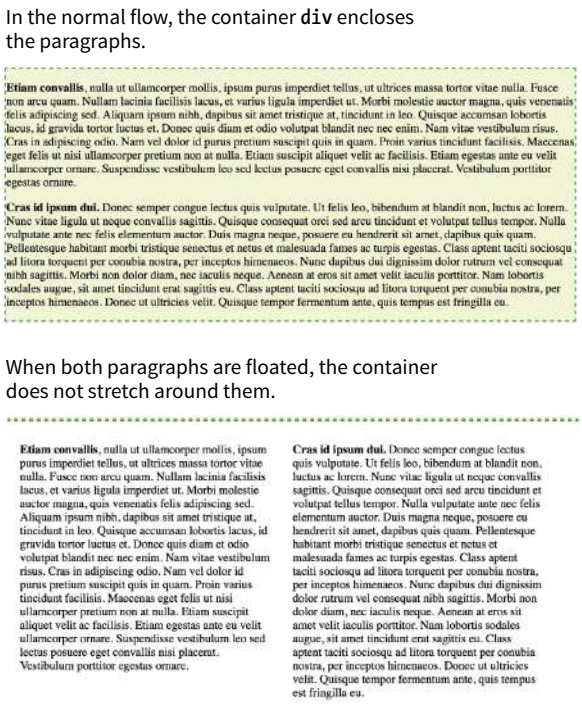

In the normal flow, the container div encloses the paragraphs.
Etiam convallis, nulla ut ullamcorper mollis, ipsum purus imperdiet tellus, ut ultrices massa tortor vitae nulla. Fusce non arcu quam. Nullam lacinia facilisis lacus, et varius ligula imperdiet ut. Morbi molestie auctor magna, quis venenatis felis adipiscing sed. Aliquam ipsum nibh, dapibus sit amet tristique at, tincidunt in leo. Quisque accumsan lobortis lacus, id gravida tortor luctus et. Donec quis diam et odio volutpat blandit nec nec enim. Nam vitae vestibulum risus. Cras in adipiscing odio. Nam vel dolor id purus pretium suscipit quis in quam. Proin varius tincidunt facilisis. Maecenas eget felis ut nisi ullamcorper pretium non at nulla. Etiam suscipit aliquet velit ac facilisis. Etiam egestas ante eu velit ullamcorper ornare. Suspendisse vestibulum leo sed lectus posuere eget convallis nisi placerat. Vestibulum porttitor egestas ornare.
Cras id ipsum dui. Donec semper congue lectus quis vulputate. Ut felis leo, bibendum at blandit non, luctus ac lorem. Nunc vitae ligula ut neque convallis sagittis. Quisque consequat orci sed arcu tincidunt et volutpat tellus tempor. Nulla vulputate ante nec felis elementum auctor. Duis magna neque, posuere eu hendrerit sit amet, dapibus quis quam. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nunc dapibus dui dignissim dolor rutrum vel consequat nibh sagittis. Morbi non dolor diam, nec iaculis neque. Aenean at eros sit amet velit iaculis porttitor. Nam lobortis sodales augue, sit amet tincidunt erat sagittis eu. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Donec ut ultricies velit. Quisque tempor fermentum ante, quis tempus est fringilla eu.
When both paragraphs are floated, the container does not stretch around them.
Etiam convallis, nulla ut ullamcorper mollis, ipsum purus imperdiet tellus, ut ultrices massa tortor vitae nulla. Fusce non arcu quam. Nullam lacinia facilisis lacus, et varius ligula imperdiet ut. Morbi molestie auctor magna, quis venenatis felis adipiscing sed. Aliquam ipsum nibh, dapibus sit amet tristique at, tincidunt in leo. Quisque accumsan lobortis lacus, id gravida tortor luctus et. Donec quis diam et odio volutpat blandit nec nec enim. Nam vitae vestibulum risus. Cras in adipiscing odio. Nam vel dolor id purus pretium suscipit quis in quam. Proin varius tincidunt facilisis. Maecenas eget felis ut nisi ullamcorper pretium non at nulla. Etiam suscipit aliquet velit ac facilisis. Etiam egestas ante eu velit ullamcorper ornare. Suspendisse vestibulum leo sed lectus posuere eget convallis nisi placerat. Vestibulum porttitor egestas ornare.
Cras id ipsum dui. Donec semper congue lectus quis vulputate. Ut felis leo, bibendum at blandit non, luctus ac lorem. Nunc vitae ligula ut neque convallis sagittis. Quisque consequat orci sed arcu tincidunt et volutpat tellus tempor. Nulla vulputate ante nec felis elementum auctor. Duis magna neque, posuere eu hendrerit sit amet, dapibus quis quam. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nunc dapibus dui dignissim dolor rutrum vel consequat nibh sagittis. Morbi non dolor diam, nec iaculis neque. Aenean at eros sit amet velit iaculis porttitor. Nam lobortis sodales augue, sit amet tincidunt erat sagittis eu. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Donec ut ultricies velit. Quisque tempor fermentum ante, quis tempus est fringilla eu.

FIGURE 16-10. The container box disappears entirely when all its contents are floated.

When both paragraphs (that is, all of the content within the `div`) are floated, as shown in the bottom part of the figure and specified by this rule), the element box for the `#container` closes up to a height of zero, leaving the floats hanging down below (you can still see the empty border at the top). There's no content left in the normal flow to give the containing `div` height. This clearly is not the effect we are after.

Fortunately, there is a simple solution to this problem (see **Note**). Setting the display type of the containing element to `flow-root` establishes it as a **block formatting context** and ensures that all of the contents of the box stay inside it:

```
#container {
  display: flow-root;
  ...
}
```

FIGURE 16-11 shows the result of applying the `flow-root` display type to the previous examples. Problem solved!

That covers the fundamentals of floating. If you're thinking that rectangular text wraps are a little *ho-hum*, in the next section you'll learn how to add some pizzazz (or just eliminate extra whitespace) by using CSS Shapes.

NOTE

If you need to support old browser versions or Internet Explorer, you will need to use the ol' "Clearfix" hack to keep containing elements open. It works by generating a block element after the containing element that clears all floats. You can read more about the technique in Thierry Koblentz's article "The Very Latest Clearfix Reloaded" (cssmojo.com/the-very-latest-clearfix-reloaded). The minimal markup for the Clearfix hack is as follows:

```
#container::after {
  content: " ";
  display: block;
  clear: both;
}
```

SUNDAE BUFFET
Topping list
Etiam convallis, nulla ut ullamcorper mollis, ipsum purus imperdiet tellus, ut ultrices massa tortor vitae nulla. Fusce non arcu quam. Nullam lacinia facilisis lacus, et varius ligula imperdiet ut. Morbi molestie auctor magna, quis venenatis felis adipiscing sed. Aliquam ipsum nibh, dapibus sit amet tristique at, tincidunt in leo. Quisque accumsan lobortis lacus, id gravida tortor luctus et. Donec quis diam et odio volutpat blandit nec nec enim. Nam vitae vestibulum risus. Cras in adipiscing odio. Nam vel dolor id purus pretium suscipit quis in quam. Proin varius tincidunt facilisis. Maecenas eget felis ut nisi ullamcorper pretium non at nulla. Etiam suscipit aliquet velit ac facilisis. Etiam egestas ante eu velit ullamcorper ornare. Suspendisse vestibulum leo sed lectus posuere eget convallis nisi placerat. Vestibulum porttitor egestas ornare.
Cras id ipsum dui. Donec semper congue lectus quis vulputate. Ut felis leo, bibendum at blandit non, luctus ac lorem. Nunc vitae ligula ut neque convallis sagittis. Quisque consequat orci sed arcu tincidunt et volutpat tellus tempor. Nulla vulputate ante nec felis elementum auctor. Duis magna neque, posuere eu hendrerit sit amet, dapibus quis quam. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Nunc dapibus dui dignissim dolor rutrum vel consequat nibh sagittis. Morbi non dolor diam, nec iaculis neque. Aenean at eros sit amet velit iaculis porttitor. Nam lobortis sodales augue, sit amet tincidunt erat sagittis eu. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos. Donec ut ultricies velit. Quisque tempor fermentum ante, quis tempus est fringilla eu.

FIGURE 16-11. Our hanging floats are now contained.

FANCY TEXT WRAP WITH CSS SHAPES

Look at the previous float examples, and you will see that the text always wraps in a rectangular shape around a floated image or element box. However, you can change the shape of the wrapped text to a circle, ellipse, polygon, or any image shape by using the `shape-outside` property. The following is a quick introduction to CSS Shapes (see **Note**), which should inspire and prepare you for more exploration on your own. You can find more detailed instruction in my "CSS Shapes" article, available at ***learningwebdesign.com/articles/***. And, of course, CSS Shapes Module Level 1 (***w3.org/TR/css-shapes-1/***) is always there as the definitive reference.

shape-outside

Values: `none` | `circle()` | `ellipse()` | `polygon()` | `inset()` | `url()` | [`margin-box` | `padding-box` | `border box` | `content-box`]

Default: none

Applies to: Floats

Inherits: No

NOTE

If you search the web for "CSS shapes," you will certainly come across that term used for a technique that uses CSS to draw geometric shapes such as triangles, arrows, circles, and so on. I introduce this topic briefly in **Chapter 26, Web Image Basics**.

FIGURE 16-12 shows the default text wrap around a floated image (left) and the same wrap with **shape-outside** applied (right). This is the kind of thing you'd expect to see in a print magazine, but now we can do it on the web!

It is worth noting that you can change the text wrap shape around any floated element, but I'll focus on images in this discussion, as text elements are generally boxes that fit nicely in the default rectangular wrap.

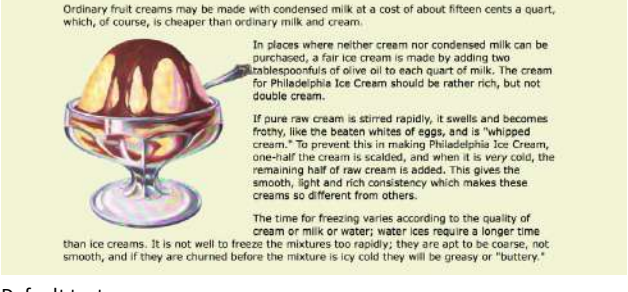
There are two approaches to making text wrap around a shape. One way is to use **url()** to specify an image that has transparent areas as the shape to wrap around. With this method, text flows into the transparent areas of the image and stops at the opaque areas (this is the technique used in FIGURE 16-12; you can learn more about it in the "CSS Shapes" article).

The other way is to provide the path coordinates of the wrap shape with **circle()**, **ellipse()**, or **polygon()**. I'll give a brief example of each here. The **circle()** notation creates a circle shape for the text to wrap around. The value provided within the parentheses represents the length of the radius (**circle(*radius*)**). This example floats an image left and wraps the text in a circle shape with a radius of 150px (FIGURE 16-13, left):

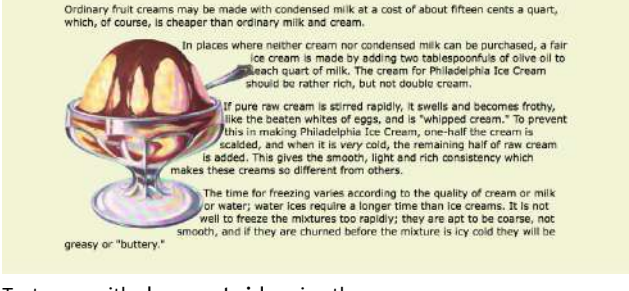
```
img.round {
  float: left;
  shape-outside: circle(150px);
}
```

Elliptical shapes are created with the **ellipse()** notation, which provides the horizontal and vertical radius lengths followed by the word **at** and then the *x* and *y* coordinates for the center of the shape (**ellipse(*rx ry* at *x y*)**). This rule creates an ellipse with a 100-pixel horizontal radius and a 150-pixel vertical radius, centered in the floated element (FIGURE 16-13, center):

```
img.round {
  float: left;
  shape-outside: ellipse(100px 150px at 50% 50%);
}
```


Ordinary fruit creams may be made with condensed milk at a cost of about fifteen cents a quart, which, of course, is cheaper than ordinary milk and cream.
In places where neither cream nor condensed milk can be purchased, a fair ice cream is made by adding two tablespoonfuls of olive oil to each quart of milk. The cream for Philadelphia Ice Cream should be rather rich, but not double cream.
If pure raw cream is stirred rapidly, it swells and becomes frothy, like the beaten whites of eggs, and is "whipped cream." To prevent this in making Philadelphia Ice Cream, one-half the cream is scalded, and when it is very cold, the remaining half of raw cream is added. This gives the smooth, light and rich consistency which makes these creams so different from others.
The time for freezing varies according to the quality of cream or milk or water; water ices require a longer time than ice creams. It is not well to freeze the mixtures too rapidly; they are apt to be coarse, not smooth, and if they are churned before the mixture is icy cold they will be greasy or "buttery."

Default text wrap


Ordinary fruit creams may be made with condensed milk at a cost of about fifteen cents a quart, which, of course, is cheaper than ordinary milk and cream.
In places where neither cream nor condensed milk can be purchased, a fair ice cream is made by adding two tablespoonfuls of olive oil to each quart of milk. The cream for Philadelphia Ice Cream should be rather rich, but not double cream.
If pure raw cream is stirred rapidly, it swells and becomes frothy, like the beaten whites of eggs, and is "whipped cream." To prevent this in making Philadelphia Ice Cream, one-half the cream is scalded, and when it is very cold, the remaining half of raw cream is added. This gives the smooth, light and rich consistency which makes these creams so different from others.
The time for freezing varies according to the quality of cream or milk or water; water ices require a longer time than ice creams. It is not well to freeze the mixtures too rapidly; they are apt to be coarse, not smooth, and if they are churned before the mixture is icy cold they will be greasy or "buttery."

Text wrap with **shape-outside** using the transparent areas of the image as a guide

FIGURE 16-12. An example of text wrapping around an image with **shape-outside**

`polygon()` lets you create a custom path using a series of comma-separated pairs of *x* and *y* coordinates along the path. This style rule creates the wrap effect shown in FIGURE 16-13, right:

```
img.wrap {
  float: left;
  width: 300px;
  height: 300px;
  shape-outside: polygon(0px 0px, 186px 0px, 225px 34px, 300px 34px,
300px 66px, 255px 88px, 267px 127px, 246px 178px, 192px 211px, 226px
236px, 226px 273px, 209px 300px, 0px 300px);
}
```

Holy coordinates! That's a lot of numbers, and my path was fairly simple. I got the coordinates by opening the image in Photoshop and gathering them manually. Firefox has a built-in CSS Shape Path Editor that you can access in the CSS Rules Panel of the developer tools (*oreil.ly/kLkLj*). Chrome offers a similar CSS Shapes Editor as a plug-in that you can download from the Chrome Web Store. Both allow you to edit shapes that are already in place (in other words, you can't add a new shape with the editors), so you may need a few coordinates to get started.

There's no better way to learn than by doing it yourself. Why don't we make the text wrap around the images in the Black Goose Bakery page in a more interesting way (EXERCISE 16-2)?

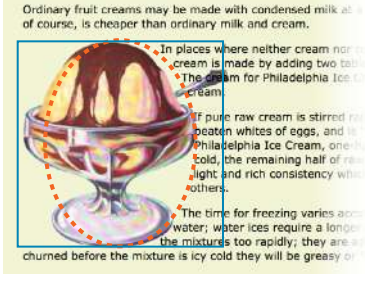
Well, that covers floating! You've learned how to float elements left and right, clear the following elements so they start below the floats, and even make fancy text wrapping shapes. Now let's move on to the other approach to moving elements around on the page—positioning.

circle()

Ice cream may be molded in the freezer; you will then
which serves very well for puddings that are to be garni
and extra expense for salt and ice.
As cold water is warmer
you lift the can or mold,
under the cold water
bottom and remove
knife, hold the mol
nine times out of
perfect shape of
and refuses to co
from warm water
and unless you kn
much melted to ga
All frozen puddings,
frozen and molded acc
The quantities given in these
that for a smaller number of persons they can be easily

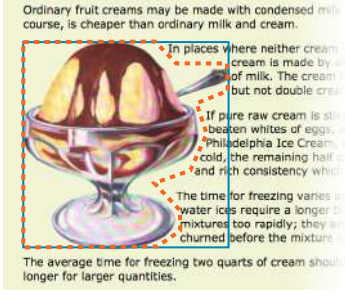
How the float with a circle wrap shape appears in the browser

ellipse()


Ordinary fruit creams may be made with condensed milk a
of course, is cheaper than ordinary milk and cream.
In places where neither cream nor
cream is made by adding two
The cream for Philadelphia Ice
cream
If pure raw cream is stirred
beaten whites of eggs, and
Philadelphia Ice Cream, one
cold, the remaining half of
light and rich consistency
others.
The time for freezing varies
water; water ices require a
the mixtures too rapidly; they
churned before the mixture is icy cold they will be greasy

The edges of the image (blue) and ellipse path (dotted orange) revealed

polygon()


Ordinary fruit creams may be made with condensed
course, is cheaper than ordinary milk and cream.
In places where neither
cream is made by
of milk. The cream
but not double
If pure raw cream is
beaten whites of eggs,
Philadelphia Ice Cream,
cold, the remaining half
and rich consistency
The time for freezing varies
water ices require a longer
mixtures too rapidly; they
churned before the mixture
The average time for freezing two quarts of cream
longer for larger quantities.

The edges of the image (blue) and polygon path (dotted orange) revealed

FIGURE 16-13. The `circle()`, `ellipse()`, and `polygon()` values of `shape-outside` applied to a left-floated image

EXERCISE 16-2. Adding shapes around floats

The bread and muffin images on the Black Goose Bakery page provide a nice opportunity to try out CSS Shapes. As you work through this exercise, if you don't see the wrap shape change, check your code, but also check your browser version—versions released prior to 2019 do not support CSS Shapes.

Open the latest version of the style sheet and look for the section labeled `/* main "products" styles */`. We'll put the image wrap styles there to keep our style sheet organized.

1. Target each image individually using an attribute selector (there is one set up for "muffin" already). Start out simply and make the text wrap around a circle. Set the radius of the circle to 125px for the bread image and 110px for the muffin:

```
img[src*="bread"] {
  shape-outside: circle(125px);
}
img[src*="muffin"] {
  margin-left: 50px;
  shape-outside: circle(110px);
}
```

2. Save the styles and take a look at the *bakery.html* page in a supporting browser. The circles look pretty good, but we could improve the wrap around the bread image by making it an ellipse. Add the new ellipse declaration after the circle declaration, so it overrides the previous style (or delete and replace the original declaration):

```
img[src*="bread"] {
  shape-outside: circle(125px);
  shape-outside: ellipse(130px 105px at 50% 50%);
}
```

3. If you're feeling ambitious, try adding a polygon wrap shape around the muffin image instead of the circle (if not, don't worry about it, nobody will judge you!). You'll need to copy these coordinates manually, or copy and paste them from the finished exercise provided with the materials for this chapter:

```
img[src*="muffin"] {
  ...
  shape-outside: polygon(0px 0px, 197px 0px, 241px
31px, 241px 68px, 226px 82px, 219px 131px, 250px
142px, 250px 158px, 0px 158px);
}
```

The final result is shown in FIGURE 16-14. It is most apparent when the browser window is sufficiently narrow that enough lines wrap to reveal the shape. For browsers that don't support shapes, the rectangular whitespace is just fine.

Fresh from the Oven
BREADS
Our breads are made daily from highest-quality whole grain flour, water, salt, and yeast or sourdough starter. Simply and naturally, and never any preservatives. Patience is key to achieving the proper level of fermentation and baking each loaf to perfection. Available in whole grain, sourdough, olive loaf, classic rye, and potato-onion.
LEARN MORE ABOUT OUR BAKING PROCESS...
MUFFINS
Every day, we offer a large selection of muffins, including blueberry, multi-berry, bran, corn, lemon- poppyseed, and chocolate. Our muffins are made from scratch each day. Stop by to see our seasonal muffin flavors!
LEARN MORE ABOUT HOW WE MAKE OUR MUFFINS...

FIGURE 16-14. The bakery page with text wrapping around images in an ellipse (bread) and polygon (muffin) using CSS Shapes

POSITIONING BASICS

CSS provides several methods for positioning elements on the page. They can be positioned relative to where they would normally appear in the flow, or removed from the flow altogether and placed at a particular spot on the page. You can also position an element relative to the viewport, and it will stay put while the rest of the page scrolls.

Types of Positioning

`position`

Values: `static` | `relative` | `absolute` | `fixed` | `sticky`

Default: `static`

Applies to: All elements

Inherits: No

The **`position`** property indicates that an element is to be positioned and specifies which positioning method to use. I'll introduce each keyword value briefly here, and then provide a more detailed look at each method in the remainder of this chapter.

`static`

This is the normal positioning scheme, in which elements are positioned as they occur in the normal document flow.

`relative`

Relative positioning moves the element box relative to its original position in the flow. The distinctive behavior of relative positioning is that the space the element would have occupied in the normal flow is preserved as empty space.

`absolute`

Absolutely positioned elements are removed from the document flow entirely and positioned with respect to the viewport or a containing element (we'll talk more about this later). Unlike with relatively positioned elements, the space they would have occupied is closed up. In fact, they have no influence at all on the layout of surrounding elements.

`fixed`

The distinguishing characteristic of **fixed positioning** is that the element stays in one position in the viewport even when the document scrolls. Fixed elements are removed from the document flow and positioned relative to the viewport rather than another element in the document.

■ TERMINOLOGY

Viewport

We'll be sticking with the more formal term *viewport* throughout the positioning discussion, but keep in mind it could be a browser window on a desktop computer, the full screen of a mobile device, or the frame of an `iframe` element from the perspective of the web page loaded in that frame. The viewport is any space that visually displays a web page.

`sticky`

Sticky positioning is a combination of relative and fixed in that the element behaves as though it is relatively positioned until it is scrolled into a specified position relative to the viewport, at which point it remains fixed (see **Warning**).

The MDN Web Docs site has this description of a potential use case:

> *Sticky positioning is commonly used for the headings in an alphabetized listing. The B heading will appear just below the items that begin with A until they are scrolled offscreen. Rather than sliding offscreen with the rest of the content, the B heading will then remain fixed to the top of the viewport until all the B items have scrolled offscreen, at which point it will be covered up by the C heading.*

WARNING

Sticky headings may improve usability when scrolling through long tables; however, some browser versions released as recently as 2021 do not support the `sticky` *position value when it is applied to table elements such as* `thead` *or* `tr`*. Use with caution.*

Each positioning method has its purpose, but absolute positioning is the most versatile. With absolute positioning, you can place an object anywhere in the viewport or within another element. It's a handy tool when used carefully and sparingly.

Specifying Position

Once you've established the positioning method, the actual position is specified with some combination of up to four **offset** properties.

`top`, `right`, `bottom`, `left`,
`inset-block-start`, `inset-inline-start`,
`inset-block-end`, `inset-inline-end`

Values:	*length* | *percentage* | `auto`
Default:	`auto`
Applies to:	Positioned elements (where `position` value is `relative`, `absolute`, or `fixed`)
Inherits:	No

The value provided for each offset property defines the distance the element should be moved *away* from that respective edge. For example, the value of **`top`** (or its flow-relative equivalent **`inset-block-start`**) defines the distance the top outer edge of the positioned element should be offset from the top edge of the browser viewport or other containing element. A positive value for **`top`** results in the element box moving *down* by that amount (see **Note**). Similarly, a positive value for **`left`** would move the positioned element to the right (toward the center of the containing block) by that amount.

NOTE

Negative values are acceptable and move the element in the opposite direction of positive values. For example, a negative value for `top` *would have the effect of moving the element up.*

Further explanations and examples of the offset properties will be provided in the discussions of each positioning method. We'll start our exploration of positioning with the fairly straightforward **`relative`** method.

RELATIVE POSITIONING

As mentioned previously, relative positioning moves an element relative to its original spot in the flow. The space it would have occupied is preserved and continues to influence the layout of surrounding content. This is easier to understand with a simple example.

When an element is relatively positioned, the space it once occupied is preserved.

Here, I've positioned an inline `em` element. A bright background color on the `em` and a border on the containing paragraph make their boundaries apparent. First I used the `position` property to set the method to `relative`, and then I used the `top` offset property to move the element 2em down from its initial position and the `left` property to move it 3em to the right. Remember, offset property values move the element *away* from the specified edge, so if you want to move something to the right, as I did here, use the `left` offset property. The results are shown in FIGURE 16-15.

```
em {
  position: relative;
  top: 2em;  /* moves element down */
  left: 3em;  /* moves element right */
  background-color: fuchsia;
}
```

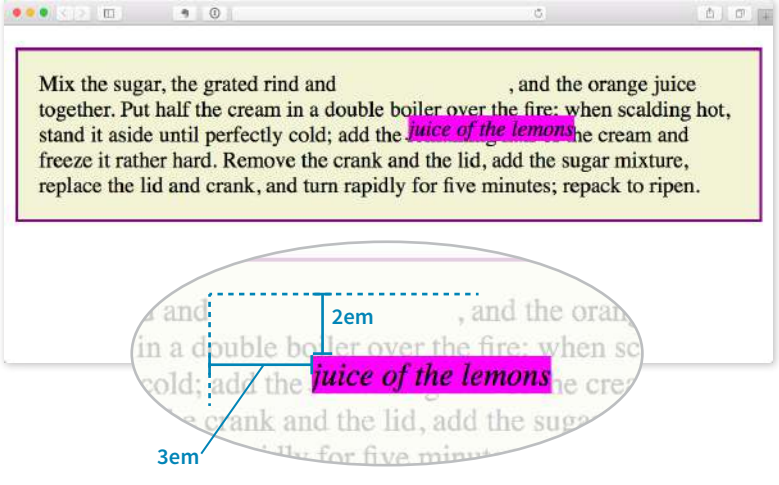

Mix the sugar, the grated rind and , and the orange juice
together. Put half the cream in a double boiler over the fire; when scalding hot,
stand it aside until perfectly cold; add the juice of the lemons he cream and
freeze it rather hard. Remove the crank and the lid, add the sugar mixture,
replace the lid and crank, and turn rapidly for five minutes; repack to ripen.
2em
3em

FIGURE 16-15. When an element is positioned with the `relative` method, the space it would have occupied is preserved.

I want to point out a few things that are happening here:

The original space in the document flow is preserved.

You can see that there is a blank space where the emphasized text would have been if the element had not been positioned. The surrounding content is laid out as though the element were still there, and therefore we say that the element still "influences" the surrounding content.

Overlap happens.

Because this is a positioned element, it can potentially overlap other elements, as happens in FIGURE 16-15.

The empty space left behind by relatively positioned objects can be a little awkward, but this is still a useful tool for nudging elements around—for example, to get an icon to line up with a button. Relative positioning is also commonly used to create a *positioning context* for an absolutely positioned element. Remember that term; I'll explain it in the next section.

ABSOLUTE POSITIONING

Absolute positioning works a bit differently and is a more flexible method for accurately placing items on the page than relative positioning. With absolute positioning, you can plop an element down right where you want it, such as a "sale" flag on a product card for an ecommerce page or an "X" button on a modal pop-up window. For demonstration purposes, I'm going to stick with the em example because it reveals some behaviors I want you to see, but this isn't the sort of thing you'd use absolute positioning for in the real world.

Now that you've seen how relative positioning works, let's see what happens when we take the same code as used in FIGURE 16-15 but set the value of the **position** property to **absolute** instead (FIGURE 16-16):

```
em {
  position: absolute;
  top: 2em;
  left: 3em;
  background-color: fuchsia;
}
```

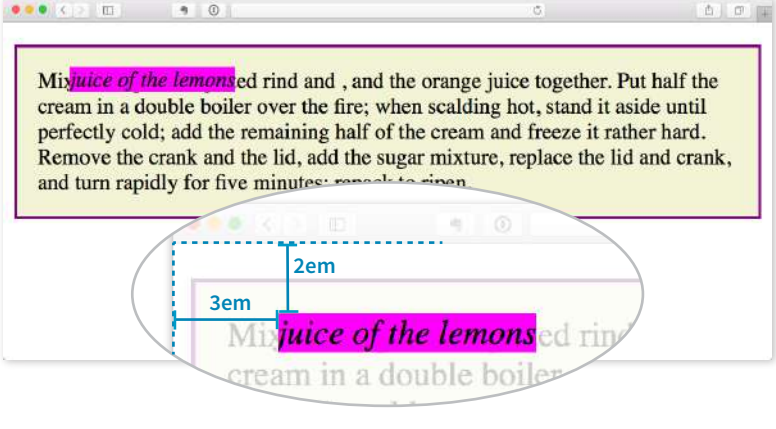

Mixjuice of the lemonsed rind and , and the orange juice together. Put half the cream in a double boiler over the fire; when scalding hot, stand it aside until perfectly cold; add the remaining half of the cream and freeze it rather hard. Remove the crank and the lid, add the sugar mixture, replace the lid and crank, and turn rapidly for five minutes;
2em
3em
juice of the lemons
cream in a double boiler

FIGURE 16-16. When an element is absolutely positioned, it is removed from the flow and the space is closed up.

As you can see in FIGURE 16-16, the space once occupied by the `em` element is now closed up. This is the case for all absolutely positioned elements. In its new position, the element box overlaps the surrounding content. In the end, absolutely positioned elements have no influence whatsoever on the layout of surrounding elements.

When an element is absolutely positioned, the space it once occupied is closed up.

The most significant difference here, however, is the location of the positioned element. This time, the offset values position the `em` element 2em down and 3em to the right of the top-left corner of the *viewport* (browser window).

But wait—before you start thinking that absolutely positioned elements are always placed relative to the viewport, I'm afraid there's more to it than that.

What actually happens in absolute positioning is that the element is positioned relative to its nearest *containing block*. It just so happens that the nearest containing block in FIGURE 16-16 is the root (`html`) element, also known as the **initial containing block**, so the offset values position the `em` element relative to the whole document.

Getting a handle on the containing block concept is the first step to tackling absolute positioning.

Containing Blocks

CSS Positioned Layout Module Level 3 states, "The position and size of an element's box(es) are sometimes computed relative to a certain rectangle, called the **containing block** of the element." It is critical to be aware of the containing block of the element you want to position. We refer to this as the **positioning context**.

The spec lays out a number of intricate rules for determining the containing block of an element, but it basically boils down to this:

- If the positioned element is *not* contained within another positioned element, then it will be placed relative to the initial containing block (created by the `html` element).
- But if the element has an ancestor (i.e., is contained within an element) that has its `position` set to `relative`, `absolute`, or `fixed`, the element will be positioned relative to the edges of *that* element instead.

FIGURE 16-16 is an example of the first case: the `p` element that contains the absolutely positioned `em` element is *not* positioned itself, and there are no other positioned elements higher in the hierarchy. Therefore, the `em` element is positioned relative to the initial containing block, which is equivalent to the viewport area.

Let's deliberately turn the `p` element into a containing block and see what happens. All we have to do is apply the `position` property to it; we don't have to actually move it. The most common way to make an element into a containing block is to set its `position` to `relative`, but not move it with

any offset values. This is what I was talking about earlier when I said that relative positioning is used to create a positioning context for an absolutely positioned element.

In this example, I'll keep the style rule for the `em` element the same, but I'll add a `position` property to the `p` element, thus making it the containing block for the positioned `em` element. FIGURE 16-17 shows the results:

```
p {
  position: relative;
  padding: 15px;
  background-color: #F2F5D5;
  border: 2px solid purple;
}
```

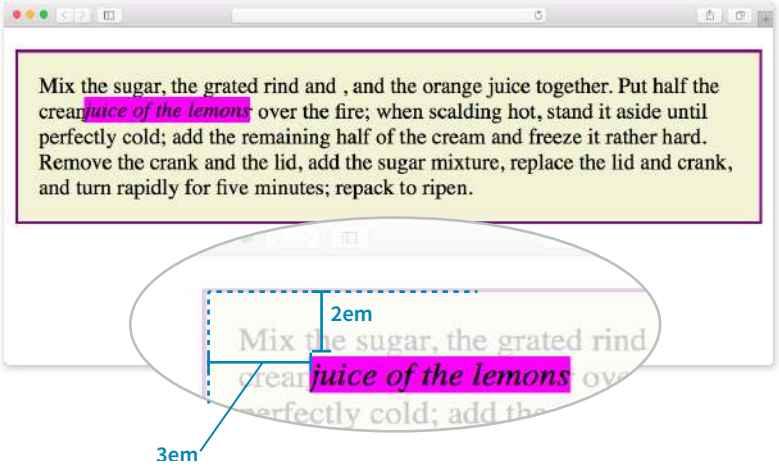

Mix the sugar, the grated rind and , and the orange juice together. Put half the cream juice of the lemons over the fire; when scalding hot, stand it aside until perfectly cold; add the remaining half of the cream and freeze it rather hard. Remove the crank and the lid, add the sugar mixture, replace the lid and crank, and turn rapidly for five minutes; repack to ripen.
2em
3em

FIGURE 16-17. The relatively positioned `p` element acts as a containing block for the `em` element.

You can see that the `em` element is now positioned 2em down and 3em to the right from the top-left corner of the paragraph box, not the browser window. Notice also that it is positioned relative to the *padding edge* of the paragraph (just inside the border), not the content area edge. This is the normal behavior when block elements are used as containing blocks (see **Note**).

NOTE

When inline elements are used as containing blocks (and they can be), the positioned element is placed relative to the content area edge, not the padding edge.

I'm going to poke around at this some more to reveal additional aspects of absolutely positioned objects. This time, I've added `width` and `margin` properties to the positioned `em` element (FIGURE 16-18):

```
em {
  width: 200px;
  margin: 25px;
  position: absolute;
  top: 2em;
  left: 3em;
  background-color: fuchsia;
}
```

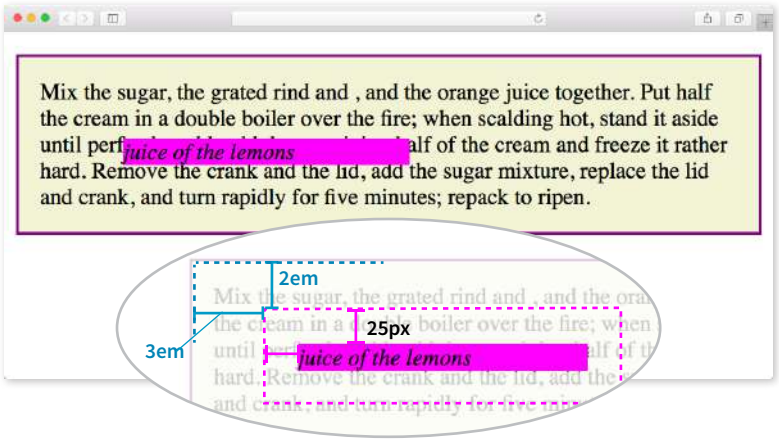

Mix the sugar, the grated rind and , and the orange juice together. Put half the cream in a double boiler over the fire; when scalding hot, stand it aside until perf
juice of the lemons
alf of the cream and freeze it rather hard. Remove the crank and the lid, add the sugar mixture, replace the lid and crank, and turn rapidly for five minutes; repack to ripen.
2em
3em
25px
juice of the lemons

FIGURE 16-18. Adding a width and margins to the positioned element

Here we can see that:

- The offset values apply to the outer edges of the element box (the outer margin edge) for absolutely positioned elements (see **Note**).
- Absolutely positioned elements always behave as block-level elements. For example, the margins on all sides are maintained, even though this is an inline element. This also permits a width to be set for the element.

NOTE

For relatively positioned elements, the offset is measured to the box itself (not the outer margin edge).

It is important to keep in mind that once you've positioned an element, it becomes the new containing block for all the elements it contains. Say you position a narrow `div` at the top-left corner of a page, creating a column. If you were to absolutely position an image within that `div` with offset values that place it in the top-right corner, it would appear in the top-right corner of that `div`, not of the entire page. Once the parent element is positioned, it acts as the containing block for the `img` and any other contained elements.

Specifying Position

Now that you have a better feel for the containing block concept, let's take some time to get better acquainted with the offset properties (see **Note**). So far, we've only seen an element moved a few ems down and to the right, but that's not all you can do, of course.

NOTE

I'll be using the physical properties (`top`, `bottom`, `left`, `right`), but they each have a flow-relative equivalent (`inset-block-start`, `inset-block-end`, `inset-inline-start`, and `inset-inline-end`, respectively).

Pixel measurements

As mentioned previously, positive offset values push the positioned element box *away* from the specified edge and toward the center of the containing block. If there is no value provided for a side, it is set to `auto`, and the browser adds enough space to make the layout work. In this example, `div#B` is contained within `div#A`, which has been given the dimensions 600 pixels wide by

300 pixels high. I've used pixel lengths for all four offset properties to place the positioned element (**div#B**) at a particular spot in its containing element (**div#A**). FIGURE 16-19 shows the result.

THE MARKUP

```
<div id="A">
  <div id="B"> </div>
</div>
```

THE STYLES

```
div#A {
  position: relative;  /* creates the containing block */
  width: 600px;
  height: 300px;
  background-color: #C6DE89;  /* green */
}

div#B {
  position: absolute;
  top: 25px;
  right: 50px;
  bottom: 75px;
  left: 100px;
  background-color: steelblue;
}
```

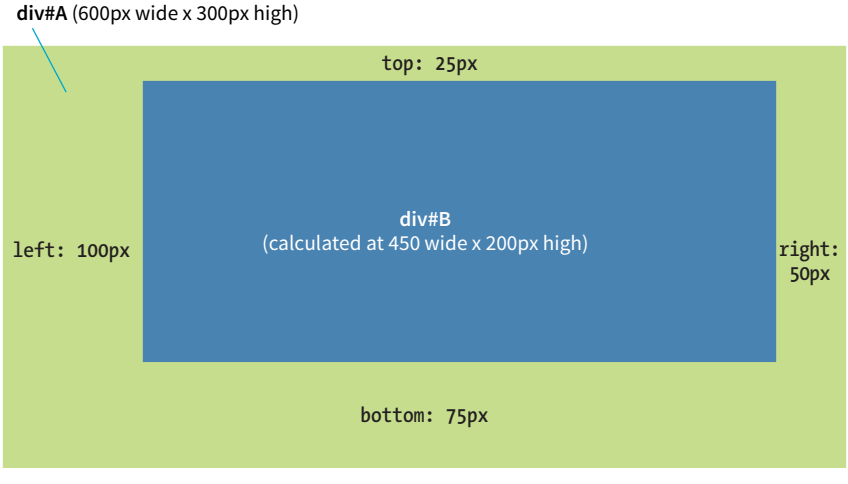


FIGURE 16-19. Setting offset values for all four sides of a positioned element

Notice that by setting offsets on all four sides, I have indirectly set the dimensions of the positioned **div#B.** It fills the 450 × 200 pixel space that is left over within the containing block after the offset values are applied. If I had also specified a width and other box properties for **div#B**, there would be a potential for conflicts if the total of the values for the positioned box and its offsets did not match the available space within the containing block.

The CSS specification provides a daunting set of rules for handling conflicts, but the upshot is that you should be careful not to over-specify box properties and offsets. In general, a width (factoring in margins as well as padding and border if you are using the **content-box** box-sizing model) and one or two offset properties are all that are necessary. Let the browser take care of the remaining calculations (see **Warning**).

Percentage values

You can also specify positions with percentage values. Here, **img#A** is positioned halfway (50%) down the left edge of the containing block (FIGURE 16-20, left), while **img#B** is positioned so that it always appears in the bottom-right corner of the containing block (FIGURE 16-20, right):

```
img#A {
  position: absolute;
  top: 50%;
  left: 0%;
}
img#B {
  position: absolute;
  bottom: 0%;
  right: 0%;
}
```

Although the examples here specify both a vertical and a horizontal offset, it is also fine to provide just one offset for a positioned element—for example, to move it left or right into a margin using either the **right** or **left** property (see **Note**).

In EXERCISE 16-3, we'll make further changes to the Black Goose Bakery home page, this time using absolute positioning.

WARNING

Be careful when positioning elements at the bottom of the initial containing block (the `html` *element). Although you may expect them to be positioned at the bottom of the whole page, browsers actually place the elements at the bottom of the browser window. The results may be unpredictable. If you want something positioned at the bottom of your page, put it in a containing block element at the end of the document source, and go from there.*

NOTE

The % symbol could be omitted for a 0 value. This essentially turns it into a 0 length but achieves an equivalent result.

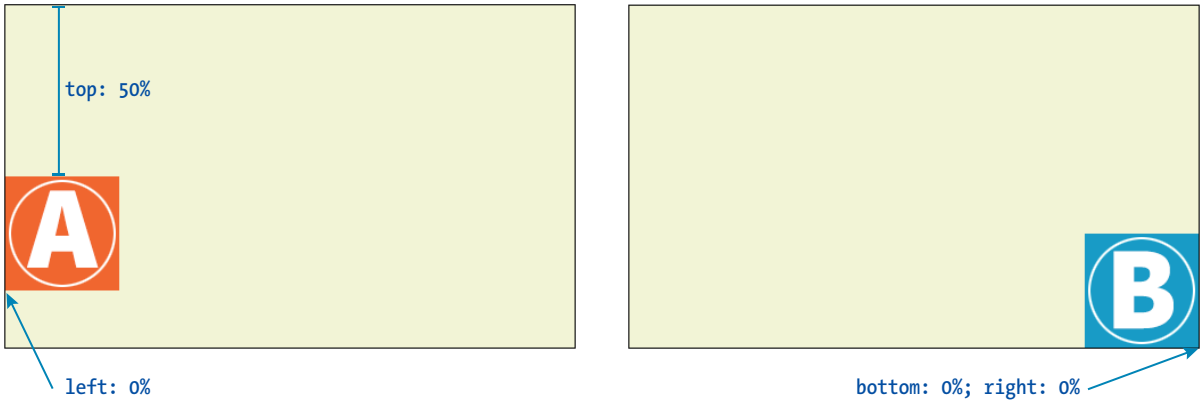


FIGURE 16-20. Using percentage values to position an element in a containing block

EXERCISE 16-3. Absolute positioning

In this exercise, we'll use absolute positioning to add an award graphic to the home page. Open the version of the *.css* file you saved in EXERCISE 16-2 in your text or code editor, along with *bakery.html*—you'll be working on both files this time.

1. Good news! Black Goose Bakery won the Farmers' Market Award, and we have the privilege of displaying an award medal on the home page. Because this is new content, we'll need to add it to the markup in *bakery.html*. Because it is nonessential information, we'll add the image in a new **div** in the **footer** of the document:

```
<footer>
  <p>All content &copy; 2024, Black Goose Bistro.</p>
  <div id="award"><img src="images/award.png" alt="Farmers' Market
Award"></div>
</footer>
```

2. Just because the award image is at the end of the source document doesn't mean it needs to display there. We can use absolute positioning to place the award in the top-left corner of the viewport by adding a new rule to the *bakery-styles.css* style sheet that positions the **div** like so (I put mine in the **/* misc styles */** section):

```
#award {
  position: absolute;
  top: 30px;
  left: 50px;
}
```

   Save both files and load the HTML document in your browser (FIGURE 16-21). Resize the browser window very narrow, and you will see that the positioned award image overlaps the header content. Notice also that when you scroll the document, the image scrolls with the rest of the page. Try playing around with other offset properties to get a feel for positioning in the viewport (or the initial containing block, to be precise).

   PS: I know that the navigation list still looks bad, but we'll fix it up in **Chapter 16**.

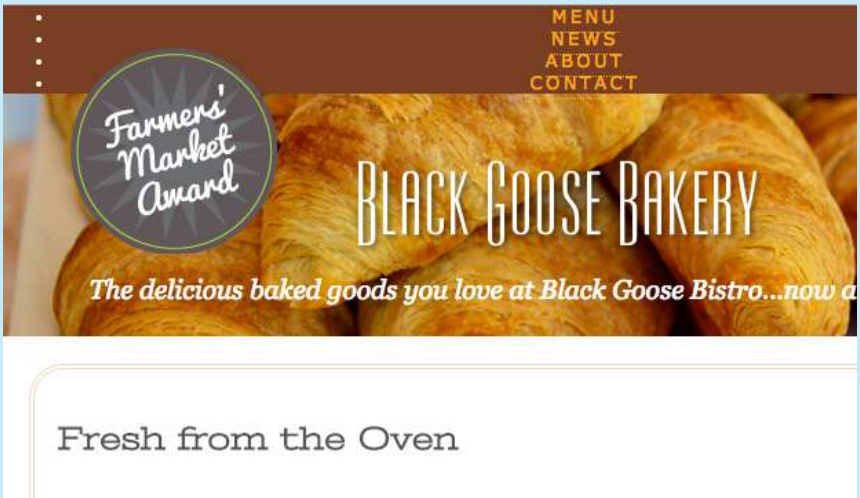


FIGURE 16-21. An absolutely positioned award graphic

FIXED POSITIONING

We've covered relative and absolute positioning, so now it's time to take on fixed positioning.

For the most part, fixed positioning works just like absolute positioning. The significant difference is that the offset values for fixed elements are *always* relative to the viewport, which means the positioned element stays put even when the rest of the page scrolls. By contrast, you may remember that when you scrolled the Black Goose Bakery page in EXERCISE 16-3, the award graphic scrolled along with the document—even though it was positioned relative to the initial containing block (equivalent to the viewport). Not so with fixed positioning, where the position is, well, *fixed*.

Fixed elements are often used for menus that should stay in the same place at the top, bottom, or side of a screen so they are always available, even when the content scrolls. Bear in mind that if you fix an element to the bottom of the viewport, you'll need to leave enough space at the end of the document so the content doesn't get hidden behind the fixed element.

Fixed elements can be problematic because they cover up other elements on the page and may take up a lot of valuable real estate on small screens (you'll see these less desirable effects in EXERCISE 16-4 if you make the browser window narrow). They are also problematic when the document is printed, so it is best to turn them off with a media query that applies only to the printed page (media queries targeting print are addressed in **Chapter 19, Responsive Web Design**).

Let's switch the award graphic on the Black Goose Bakery page to a fixed position in EXERCISE 16-4 to see the difference.

EXERCISE 16-4. Fixed positioning

This should be simple. Open the style sheet as you left it in EXERCISE 16-3 and edit the style rule for the `#award div` to make its position `fixed` rather than `absolute`:

```
#award {
  position: fixed;
  top: 30px;
  left: 50px;
}
```

Save the styles and open the page in a browser. When you scroll the page, you will see that the award graphic now stays put where you positioned it in the browser window (FIGURE 16-22). You can see that fixed positioned elements have the potential to hide content as the page scrolls. Test well to see the potential pitfalls and weigh them against the benefits.

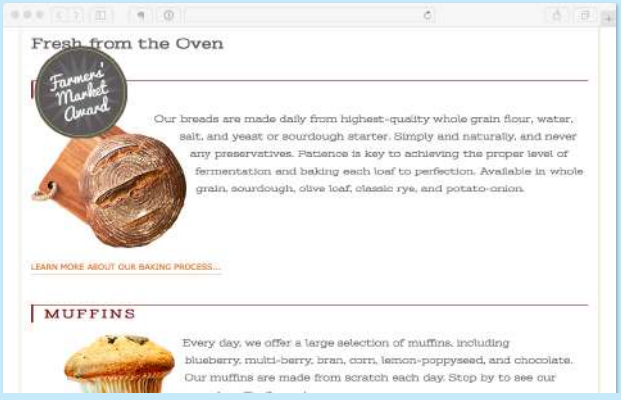

Fresh from the Oven
Farmers' Market Award
Our breads are made daily from highest-quality whole grain flour, water, salt, and yeast or sourdough starter. Simply and naturally, and never any preservatives. Patience is key to achieving the proper level of fermentation and baking each loaf to perfection. Available in whole grain, sourdough, olive loaf, classic rye, and potato-onion.
LEARN MORE ABOUT OUR BAKING PROCESS...
MUFFINS
Every day, we offer a large selection of muffins, including blueberry, multi-berry, bran, corn, lemon-poppyseed, and chocolate. Our muffins are made from scratch each day. Stop by to see our

FIGURE 16-22. The award stays in the same place in the top-left corner of the browser window when the document scrolls.

STACKING ORDER

Before I close the book on positioning, there is one last related concept that I want to introduce. As you've seen, positioned elements overlap other elements, so it follows that multiple positioned elements have the potential to stack up on one another.

By default, elements stack up in the order in which they appear in the document, but you can change the stacking order with the **z-index** property (see **Note**). Picture the **z-axis** as a line that runs perpendicular to the page, as though from the tip of your nose, through this page, and out the other side.

NOTE

The **z-index** *property is also useful for items in a grid layout, which also have the potential to overlap, as discussed in* **Chapter 18**.

z-index

Values: *number* | auto

Default: auto

Applies to: Positioned elements

Inherits: No

The value of the **z-index** property is a number (positive or negative). The higher the number, the higher the element will appear in the stack (that is, closer to your nose). Lower numbers and negative values move the element lower in the stack. Let's look at an example to make this clearer (FIGURE 16-23).

By default, elements later in the document source order stack on top of preceding elements.

You can change the stacking order with the z-index property. Higher values stack on top of lower values.

FIGURE 16-23. Changing the stacking order with the **z-index** property

In this figure there are three paragraph elements, each containing a letter image (A, B, and C, respectively), that have been absolutely positioned in such a way that they overlap on the page. By default, paragraph C will appear on top because it appears last in the source. However, by assigning higher **z-index** values to paragraphs A and B, we can force them to stack in our preferred order.

Note that the values of **z-index** do not need to be sequential, and they do not relate to anything in particular. All that matters is that higher number values position the element higher in the stack.

THE MARKUP

```
<p id="A"><img src="A.gif" alt="A"></p>

<p id="B"><img src="B.gif" alt="B"></p>

<p id="C"><img src="C.gif" alt="C"></p>
```

THE STYLES

```
#A {
  z-index: 100;
  position: absolute;
  top: 175px;
  left: 200px;
}
#B {
  z-index: 5;
  position: absolute;
  top: 275px;
  left: 100px;
}
#C {
  z-index: 1;
  position: absolute;
  top: 325px;
  left: 250px;
}
```

To be honest, the **z-index** property is not required for most page layouts, but you should know it's there if you need it. If you want to guarantee that a positioned element always ends up on top, assign it a very high **z-index** value, such as 100. If you want to make sure it's at the bottom, give it a negative value. The number itself doesn't actually matter, but be careful with **z-index** because it can easily become unwieldy as override values climb to extremely high values (e.g., 99999999999999 and beyond).

That does it for floating and positioning. In the next two chapters, you'll learn about Flexbox and Grid Layout, which are powerful tools for designing the overall structure of a page and specific page features. But first, try your hand at a few questions about floating and positioning.

TEST YOURSELF

Take a moment to see how well you absorbed the principles in this chapter. You'll find the answers in **Appendix A**.

1. Which of the following is *not* true of floated elements?
   a. All floated elements behave as block elements.
   b. Floats are positioned against the padding edge of the containing element.
   c. The content of elements flows around a float, but the element box is unchanged.
   d. You must provide a `width` property for floated block elements.

2. Which of these style rules is incorrect? Why?
   a. `img { float: left; margin: 20px;}`
   b. `img { float: right; width: 120px; height: 80px; }`
   c. `img { float: right; right: 30px; }`
   d. `img { float: left; margin-bottom: 2em; }`

3. How do you make sure a `footer` element always starts below any floated sidebars on the page?

4. Write the name of the positioning method or methods (static, relative, absolute, or fixed) that best matches each of the following descriptions.
   a. Positions the element relative to a containing block.
   b. Removes the element from the normal flow.
   c. Always positions the element relative to the viewport.
   d. The positioned element may overlap other content.
   e. Positions the element in the normal flow.
   f. The space the element would have occupied in the normal flow is preserved.
   g. The space the element would have occupied in the normal flow is closed up.
   h. You can change the stacking order with `z-index`.
   i. Positions the element relative to its original position in the normal flow.

CSS REVIEW: FLOATING AND POSITIONING PROPERTIES

Here is a summary of the properties covered in this chapter.

Property	Description
`clear`	Prevents an element from being laid out next to a float
`float`	Moves the element to the right or left and allows the following text to flow around it
`position`	Specifies the positioning method to be applied
`top, bottom, right, left, inset-block-start, inset-block-end, inset-inline-start, inset-inline-end`	Specify the offset amount from each respective edge
`shape-outside`	Causes content to wrap around a shape instead of the float's bounding box
`z-index`	Specifies the order of appearance within a stack of overlapping positioned elements

CHAPTER
17

GETTING IN LINE WITH FLEXBOX

So far, we've seen how to move elements around using text alignment, floats, and positioning. For decades those were our only options, and although we came up with hacks to make them do what we want, web designers were ready for a better solution. Enter the CSS Flexible Box Layout Module, or Flexbox for short.

Put simply, Flexbox is used for arranging elements into a row or column (see **Note**), but it also gives us more fine-tuned control over alignment and solves some of the frustrating limitations of our previous layout toolkit. One of the great marvels of the Flexbox model is that items resize, or **flex** to use the formal term, to fit the available space. That means it allows items to stretch or shrink inside their containers, preventing wasted space and overflow—a real plus for making layouts fit a variety of viewport sizes. Other advantages include the following:

- The ability to make all neighboring items the same height
- Easy horizontal and vertical centering (curiously elusive with old CSS methods)
- The ability to change the order in which items display, independent of their order in the source

Flexbox is good for page elements that need to line up in some manner. Practical applications include navigation bars, galleries, product listings, and form elements, as shown in FIGURE 17–1.

The Flexbox layout model is incredibly robust, but because it was designed for maximum flexibility, it may take a little time to wrap your head around it. Here's one helpful way to think about it as we head into this chapter: when you tell an element to become a flexbox, all of its child elements line up on one axis, like beads on a string. The string may be horizontal, it may hang

IN THIS CHAPTER

Flex containers and items

Flow direction and wrapping

Flex item alignment

Controlling item "flex"

NOTE

To arrange elements into rows and columns, there is CSS Grid Layout, which I'll cover in **Chapter 18, Grid Layout**.

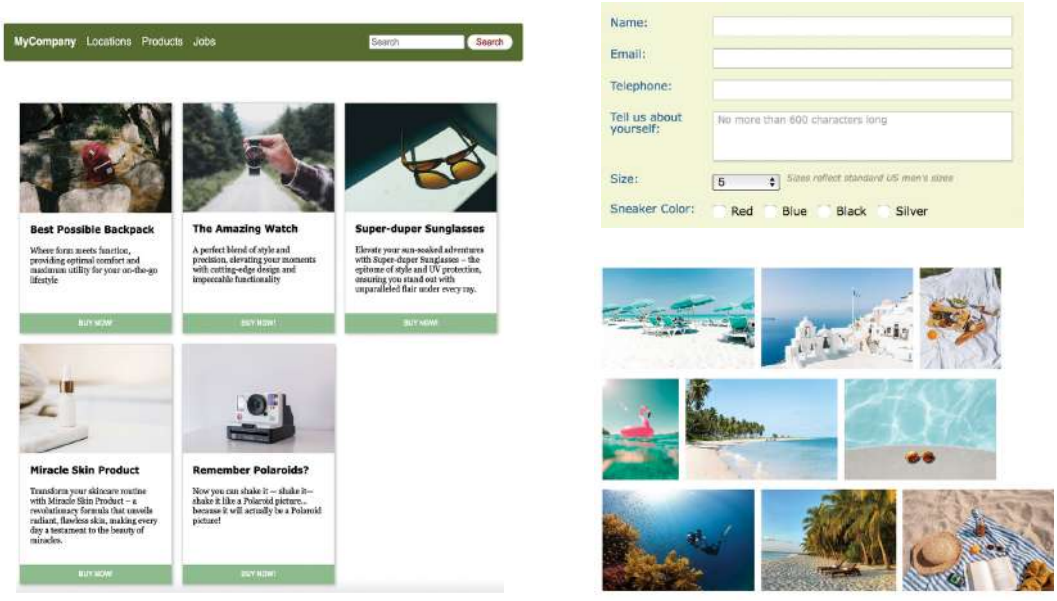


FIGURE 17-1. CSS Flexbox is a good option for aligning navigation bars, forms, product listings, image galleries, and much more.

Browser Support for Flexbox

As of this writing, 98% of browsers in use support the current Flexbox standard (Flexible Box Layout Module Level 1, *w3.org/TR/css-flexbox-1/*), so it is a very reliable CSS feature.

That said, it's worth knowing that the spec went through some significant changes on its path to standardization. Many browsers prior to 2015 supported Flexbox only with their respective vendor prefixes (I talk about prefixes in **Chapter 21, More CSS Techniques**). Prior to 2012, browsers supported the initial Flexbox draft, which differed significantly from the one we use now.

It is unlikely you will need to support browsers more than a decade old, or the last versions of Internet Explorer. If you do, be sure to brush up on workarounds like feature detection (also covered in **Chapter 21**).

vertically, or it may even wrap onto multiple lines, but the beads are always on one string (or to use the proper term, one **axis**).

In this chapter, you'll learn how to:

- Set up a flex container.
- Control the direction the flex items flow and whether they may wrap onto new lines.
- Set the alignment of items in the flexbox.
- Control whether and how the items shrink or expand to fill the available space.
- Change the order in which items appear in the lineup.

We're going to start by looking at the properties related to the flex container (direction, wrapping, and alignment), then we'll move on to the properties for the child flex items, including how items may shrink or grow and in what order they appear.

SETTING UP A FLEX CONTAINER

You've already learned about the block layout display mode for stacking elements in the normal flow and the inline mode for displaying content within it horizontally. Flexbox is another display mode with its own behaviors. To

turn on Flexbox mode for an element, set its **display** property to **flex** or **inline-flex** (see **Note**). It is now a **flex container**, and all of its direct child elements (whether they are **div**s, list items, paragraphs, etc.) *automatically* behave as **flex items** in that container. The flex items are laid out and aligned along the axis, which may appear on one line or wrap onto multiple lines.

FIGURE 17-2 shows the effect of simply adding **display: flex** to a **div**, thus turning on the Flexbox switch. I've added a blue border to the container to make its boundaries clear. To save space, I am not showing purely cosmetic styles such as colors and fonts. Also keep in mind that although I'm using simple numbered boxes here to show the effects of the code, these could be any page elements, such as those you saw in FIGURE 17-1.

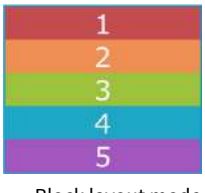
THE MARKUP

```
<div id="container">
  <div class="box box1">1</div>
  <div class="box box2">2</div>
  <div class="box box3">3</div>
  <div class="box box4">4</div>
  <div class="box box5">5</div>
</div>
```

THE STYLES

```
#container {
  display: flex;
}
```

By default, the **div**s display as block elements, stacking up vertically. Turning on Flexbox mode makes them line up in a row.



Block layout mode

`display: flex;`

Flexbox layout mode

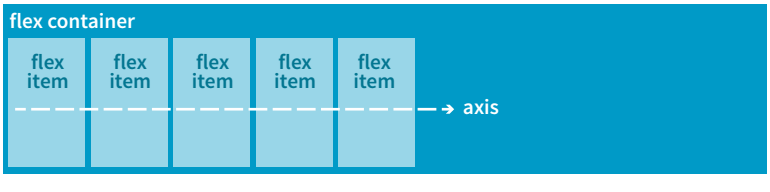


FIGURE 17-2. Applying the **flex** display mode to the container turns its child elements into flex items that line up along one axis. You don't need to do anything to the child elements themselves.

NOTE

The **inline-flex** *value generates an inline-level flex container box. I'll be focusing on the more commonly used* **flex** *value in this chapter.*

You can see that the items have lined up in a row from left to right, which is the default Flexbox behavior if your page is in English or another language written in rows from left to right. That is because the default Flexbox direction matches the direction of the language the page is written in. It would go from right to left by default in Hebrew or Arabic, or in columns if the page is written in a vertical writing system.

It is worth noting that you can turn any flex item into a flex container by setting its `display` property to `flex`, resulting in a **nested flexbox**. In fact, you'll get to try that yourself in an upcoming exercise. Some Flexbox solutions use flexboxes nested several layers deep.

FLEXBOX FUN FACTS

Here are a few things to know about Flexbox and flex item behavior:

- `float`, `clear`, multicolumn layout, and `vertical-align` do not work with elements in Flexbox mode.
- Margins do not collapse in Flexbox mode. The margin edges of items are placed at the start or end of the flex line and do not overlap the padding of the container. The margins on neighboring items add up.
- The spec recommends avoiding percentage values for margins and padding on flex items because of unpredictable results.

CONTROLLING THE "FLOW" WITHIN THE CONTAINER

Once you turn an element into a flex container, there are a few properties you can set on that container to control how items flow within it. The **flow** refers to the direction in which flex items are laid out as well as whether they are permitted to wrap onto additional lines.

Specifying Flow Direction

You may be happy with items lining up in a row by default, as shown in FIGURE 17-2, but there are a few other options that are controlled with the `flex-direction` property.

`flex-direction`

Values: `row` | `column` | `row-reverse` | `column-reverse`

Default: `row`

Applies to: Flex containers

Inherits: No

The default value is `row`, as we saw in the previous example (see the "**Row and Column Direction**" sidebar for important clarification). You can also specify that items get aligned vertically in a `column`. The other options, `row-reverse` and `column-reverse`, also arrange items horizontally and vertically, as you would expect, but they start filling from the opposite end. FIGURE 17-3 shows the effects of each keyword as applied to our simple example.

Row and Column Direction

In writing systems with horizontal lines of text, the `row` keyword lays items out horizontally, which is how we Westerners typically think of a "row" being oriented. Bear in mind that in vertically oriented languages, `row` aligns items vertically, in keeping with the default direction of the writing system. Similarly, `column` results in horizontally aligned items in vertical languages.

This is a behavior worth knowing about; however, because we are creating English language sites in this book, I'll be following the assumption that row = horizontal and column = vertical throughout this chapter for simplicity's sake.

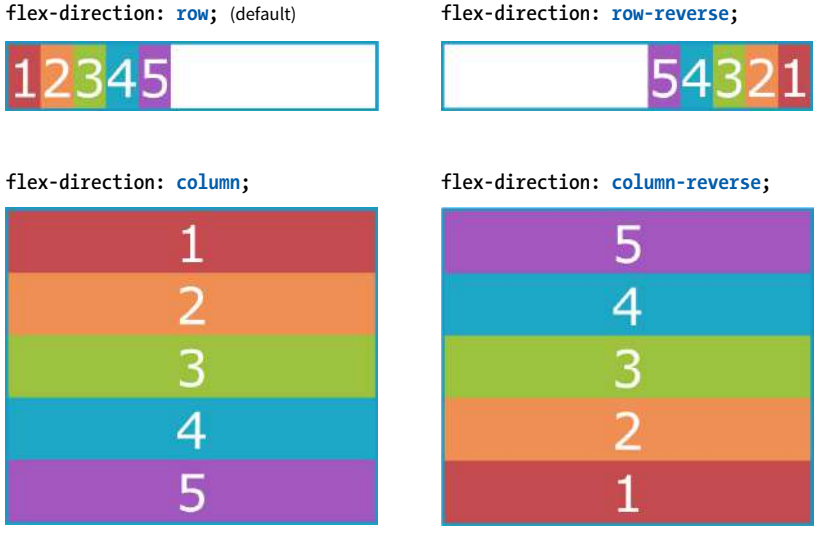


FIGURE 17-3. Examples of `flex-direction` values `row`, `row-reverse`, `column`, and `column-reverse`

Now that you've seen Flexbox in action, it's a good time to familiarize yourself with the formal Flexbox terminology. Because the system is direction-agnostic, there are no references to physical locations like "left," "right," "top," or "bottom" in the property values. Instead, we talk about the **main axis** and the **cross axis**. The main axis is the flow direction you've specified for the flex container. For primarily horizontal languages, when `flex-direction` is set to `row`, the main axis is horizontal; for `column`, the main axis is vertical (again, rows and columns are language dependent). The cross axis is whatever

direction is perpendicular to the main axis. The parts of a flex container are illustrated in FIGURE 17-4.

The main axis is the flow direction you've specified for the flex container. The cross axis is perpendicular to the main axis.

In addition to the axes, understanding the other parts of the Flexbox system makes the properties easier to learn. Both the main and cross axes have a start and an end, based on the direction in which the items flow. Note that the Flexbox spec makes use of the logical terms "start" and "end" so it works for all writing directions. The **main size** is the width (or height if it's a column) of the container along the main axis, and the **cross size** is the height (or width if it's a column) along the cross axis.

Wrapping onto Multiple Lines

If you have a large or unknown number of flex items in a container and don't want them to all just get squished into the available space, you can allow them to break onto additional lines with the `flex-wrap` property.

■ DON'T WORRY

Keeping the main and cross axes straight as you switch between rows and columns can feel like mental gymnastics and is one of the trickier things about using Flexbox. With practice, you'll get used to it.

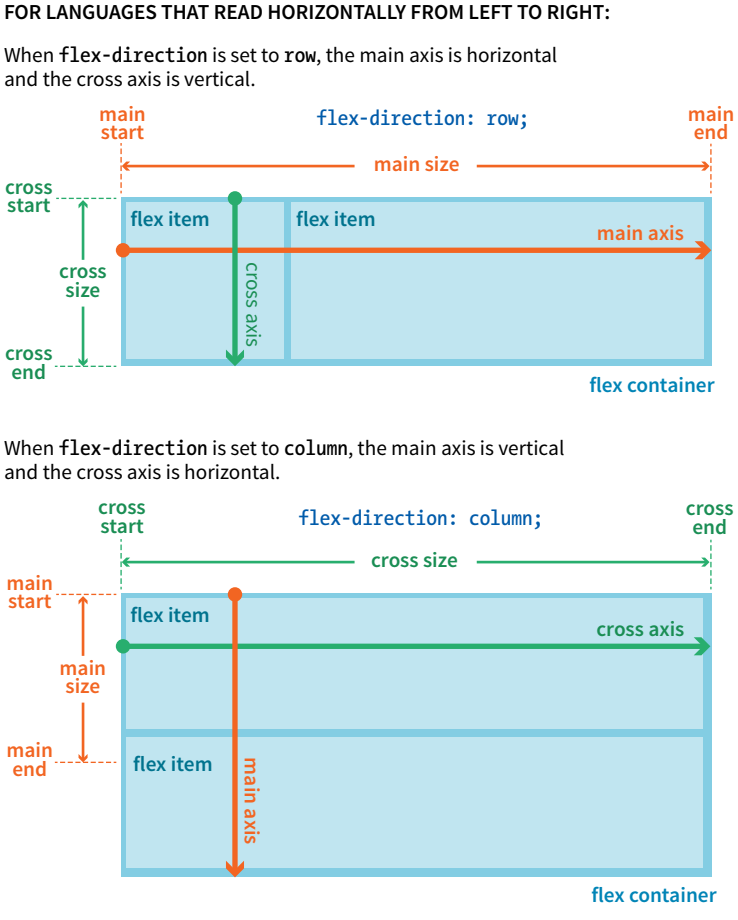


FIGURE 17-4. The parts of a flex container

flex-wrap

Values: nowrap | wrap | wrap-reverse
Default: nowrap
Applies to: Flex containers
Inherits: No

By default, items do the squish thing and do not wrap onto additional lines (**nowrap**). The **wrap** keyword turns on the ability to wrap onto multiple lines. New lines are added in the direction from *cross start* to *cross end*. For example, with the default **flex-direction** of **row**, new lines are positioned from the top down, which is what you'd expect. (Refer to FIGURE 17-4 if this terminology is still unclear.)

wrap-reverse breaks the elements onto multiple lines but flows them in the opposite direction, from cross end to cross start (from the bottom up, in this case). It feels a little esoteric, but you never know when an occasion might arise to put it to use.

I've added more **div**s to our numbered flexbox example and given the flex items a width of 25% so that only four will fit across the width of the container. FIGURE 17-5 shows a comparison of the various wrap options when the flex direction is **row**.

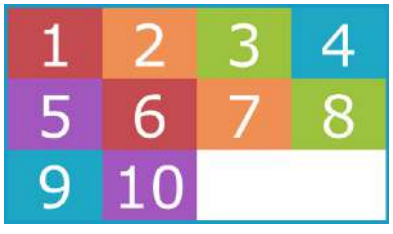
THE MARKUP

```
<div id="container">
  <div class="box box1">1</div>
  <!-- 8 more boxes here -->
  <div class="box box10">10</div>
</div>
```

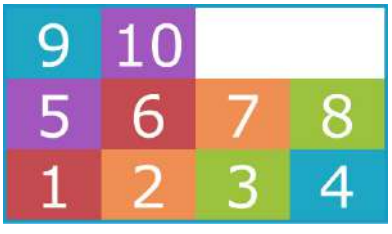
THE STYLES

```
#container {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
}
.box {
  width: 25%;
}
```

flex-wrap: wrap;



flex-wrap: wrap-reverse;

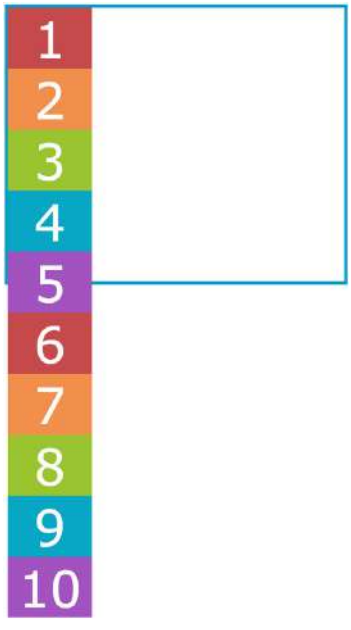


flex-wrap: nowrap; (default)

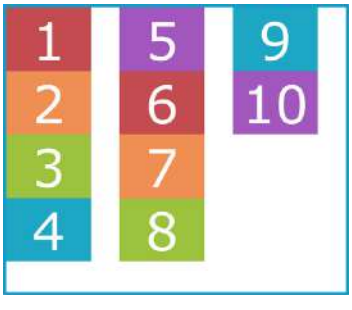
When wrapping is disabled, flex items squish to fit the viewport. If there is not enough room, items may get cut off.

FIGURE 17-5. Comparing the effects of the **nowrap**, **wrap**, and **wrap-reverse** keywords for **flex-wrap**

`flex-wrap: nowrap;` (default)



`flex-wrap: wrap;`



`flex-wrap: wrap-reverse;`

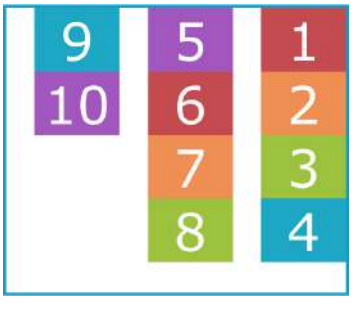


FIGURE 17-6. Comparing `nowrap`, `wrap`, and `wrap-reverse` when the items are in a column

By default, when **`flex-direction`** is set to **`column`**, the container expands to contain the height of the items. In order to see wrapping kick in, you need to set a height on the container, as I've done in the following example. FIGURE 17-6 shows how wrapping works for each of the **`flex-wrap`** keywords when **`flex-direction`** is set to **`column`**. Notice that the items are still 25% the width of their parent container, so there is space left over between the columns.

```
#container {
  display: flex;
  height: 350px;
  flex-direction: column;
  flex-wrap: wrap;
}
.box {
  width: 25%;
}
```

Putting It Together with flex-flow

The shorthand property **`flex-flow`** makes specifying **`flex-direction`** and **`flex-wrap`** short and sweet. Omitting one value results in the default value for its respective property, which means you can use **`flex-flow`** for either or both direction and wrap.

`flex-flow`

Values: *flex-direction flex-wrap*

Default: `row nowrap`

Applies to: Flex containers

Inherits: No

Using **`flex-flow`**, I could shorten the previous example (FIGURE 17-6) like so:

```
#container {
  display: flex;
  height: 350px;
  flex-flow: column wrap;
}
```

You've only scratched the surface of Flexbox, but you've got what it takes to whip that ugly **`nav`** menu on the bakery page into shape in EXERCISE 17-1.

CONTROLLING THE ALIGNMENT OF FLEX ITEMS

So far, you've seen how to turn Flexbox mode on, turning an element into a flex container and its children into flex items. You've also learned how to change the direction in which items flow and allow them to wrap onto multiple lines. The remaining container properties affect the alignment of items along the main axis (**`justify-content`**) and cross axis (**`align-items`**

EXERCISE 17-1. Making a navigation bar with Flexbox

This exercise continues work on the Black Goose Bakery site from the previous chapter. Open your most recent version of the style sheet in a text or code editor—or, if you need a fresh start, you will find an updated copy of *bakery-styles.css* with the materials for this chapter.

1. Start by removing the spacing and markers the browser applies to the **ul** element in the **nav** element. This gives us a neutral starting point for styling:

   ```
   nav ul {
     margin: 0;
     padding: 0;
     list-style-type: none;
   }
   ```

   Turn that **ul** element into a flexbox by setting its display to **flex**. As a result, all of the li elements become flex items. Because we want rows and no wrapping, the default values for **flex-direction** and **flex-wrap** are fine, so those properties may be omitted:

   ```
   nav ul {
     ...
     display: flex;
   }
   ```

   Save the document and open *bakery.html* in a browser. You should see that the navigation links are lined up tightly in a row, which is an improvement, but we have more work to do.

2. Now we can work on the appearance of the links. Create a new style rule in the **/* nav styles */** section for the a elements in the **nav** list items. Start by making them display as block elements instead of inline, then give them 1px rounded borders. Add padding within the borders (.5em top and bottom, 1em left and right) and .5em margins to give them space and to open up the brown navigation bar:

   ```
   nav ul li a {
     display: block;
     border: 1px solid;
     border-radius: 6px;
     padding: .5em 1em;
     margin: .5em;
   }
   ```

3. I want the navigation menu to be centered in the width of the **nav** section. I'm getting a little ahead of myself here because we haven't seen alignment properties yet, but this one is intuitive. Consider it a preview of what's coming up in the next section. Add the following **justify-content** declaration for the **nav ul** element:

   ```
   nav ul {
     ...
     display: flex;
     justify-content: center;
   }
   ```

FIGURE 17-7 shows the way your navigation menu should look when you are finished.

Note: We'll be using this version of the bakery site as the starting point for EXERCISE 18-5, so save it and keep it for the next chapter.

FIGURE 17-7. The list of links is now styled as a horizontal menu bar.

and **align-content**). See the sidebar **"The Box Alignment Module"** for a bit of background information on how the W3C is thinking about these alignment properties.

The Box Alignment Module

The W3C has created a separate module documenting all of the features related to the alignment of boxes within their containers. CSS Box Alignment Module Level 3 defines the concepts of main and cross axes and specifies six box alignment properties that apply to Flexbox as well as the block, grid, and table layout models. In addition, it introduces properties that control the amount of gap between rows and columns that may be applied to flex, grid, and multicolumn layouts. I'll cover the concept of "gap" in more detail in **Chapter 18**.

For now, the alignment properties are also included in the Flexbox module, but the long-term plan is to use the Box Alignment module as the definitive reference. Once it is completed, it will supersede the properties specific to Flexbox. If you'd like to get insight into the broader strategy for aligning elements within their containers, take a look at the spec at *w3.org/TR/css-align-3/*.

Alignment on the Main Axis

By default, flex items are just as wide as they need to be to contain the element's content, which means the container may have space to spare on the flex line. We saw this back in FIGURE 17-2. Also by default, the items flow in right next to each other from the "main start" (based on language direction and the direction of the flex line).

The **justify-content** property defines how extra space should be distributed around or between items that are inflexible or have reached their maximum size (see **Note**). The setting is applied *after* margins have been calculated on items and after the "flex" values have been accounted for. If items are specified to grow to fill the container width, then **justify-content** is not applicable.

justify-content

Values:	flex-start | flex-end | center | space-between | space-around
Default:	flex-start
Applies to:	Flex containers
Inherits:	No

You apply **justify-content** to the flex container element because it controls spacing within the container itself:

```
#container {
  display: flex;
  justify-content: flex-start;
}
```

As you'd expect, **flex-start** and **flex-end** position the line of items toward the start and end of the main axis, respectively, and **center** centers them.

space-between and **space-around** warrant a little more explanation. When **justify-content** is set to **space-between**, the first item is positioned at the start point, the last item goes at the end point, and the remaining space is distributed evenly between the remaining items. The **space-around** property adds an equal amount of space on the left and right side of each item, resulting in a doubling up of space between neighboring items. FIGURE 17-8 shows how items align using each of the keyword values for **justify-content**.

When the flex container's direction is set to **column** with a vertical main axis, the keywords work the same way; however, there needs to be an explicit container height with space left over in order for you to see the effect. I've changed the size of the text and set a height on the container element in FIGURE 17-9 to demonstrate the same keywords as applied to a vertical main axis when **flex-direction** is set to **column**.

NOTE

You can also distribute extra space along the main axis by making the flex items themselves grow wider to fill the available space. That is the job of the **flex** *properties, which we'll look at in a moment.*

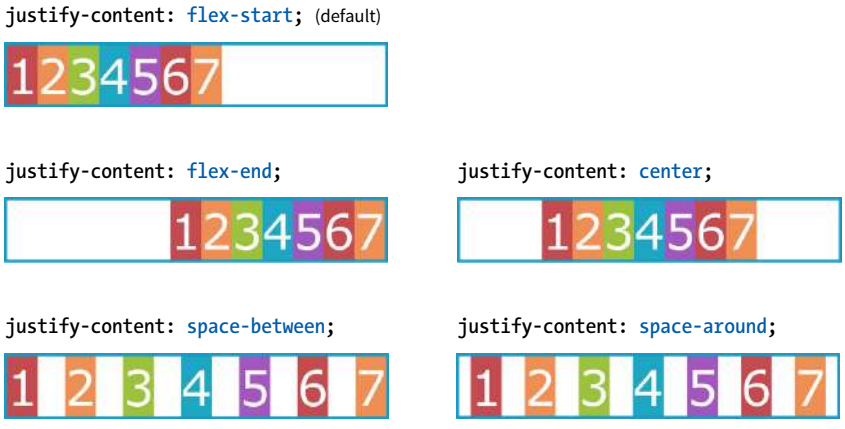


FIGURE 17-8. Options for aligning items along the main axis with `justify-content`

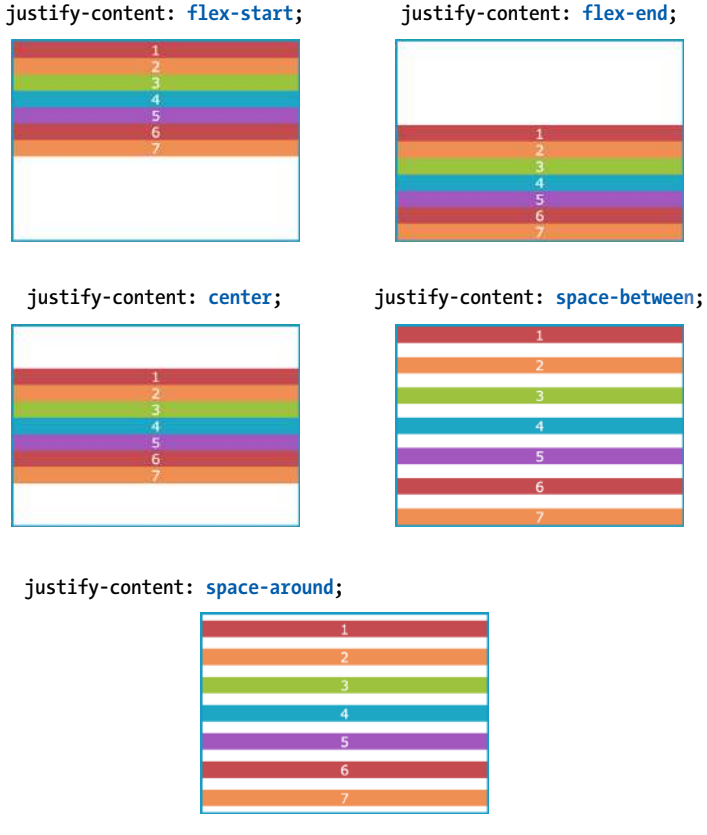


FIGURE 17-9. Options for aligning items along a vertical main axis (`flex-direction` set to `column`) with `justify-content`

Alignment on the Cross Axis

That takes care of arranging things on the main axis, but you may also want to play around with alignment on the cross axis (up and down when the direction is **row**, left and right if the direction is **column**). Cross-axis alignment and stretching are the job of the **align-items** property.

align-items

Values: flex-start | flex-end | center | baseline | stretch

Default: stretch

Applies to: Flex containers

Inherits: No

I've demonstrated the various keyword values for **align-items** as they apply to rows in FIGURE 17-10. In order to see the effect, you must specify a height for the container; otherwise, it expands just enough to contain the content with no extra space to spare. I've given the container a fixed height here to show how the items are positioned on the cross axis (i.e., vertically in relation to the container) with the different values.

Like **justify-content**, the **align-items** property applies to the flex container (that can be a little confusing because "items" is in the name):

```
#container {
  display: flex;
  flex-direction: row;
  height: 200px;
  align-items: flex-start;
}
```

The **flex-start**, **flex-end**, and **center** values should be familiar, only this time they refer to the start, end, and center of the *cross* axis. The **baseline** value aligns the baselines of the first lines of text, regardless of their size. It may be a good option for lining up elements with different text sizes, such

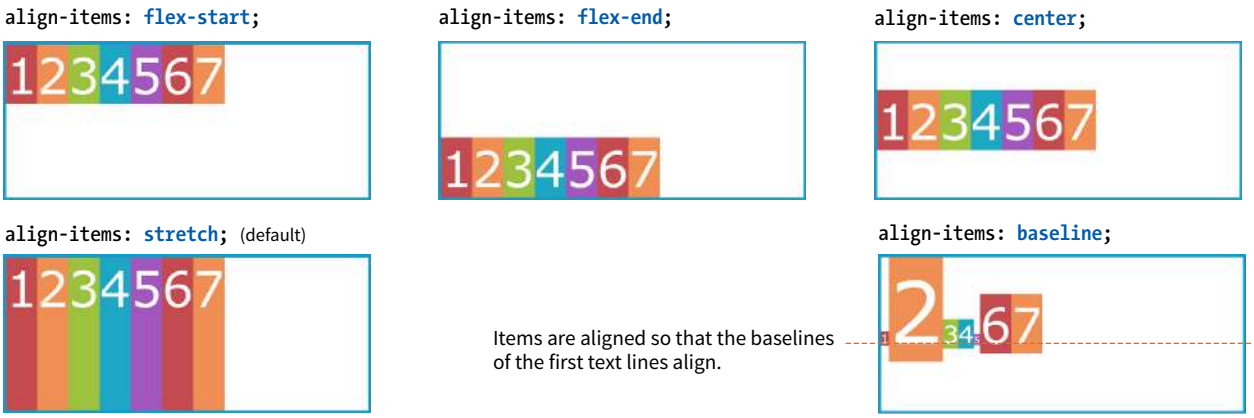


FIGURE 17-10. Options for aligning items along the cross axis with **align-items**

as headlines and paragraphs across multiple items. Finally, **stretch**, which is the default, causes items to stretch until they fill the cross axis (i.e., the height of the container).

When the flex container's direction is set to **column**, **align-items** aligns items horizontally in the container. Look back at FIGURES 17-3 and 17-9, and you will see that when I set the items in a column and did not provide any alignment information, each item stretched to the full width of the cross axis, because **stretch** is the default value.

If you'd like one or more specific items to override the cross-axis setting, use the **align-self** property on the individual item element(s). This is the first property we've seen that applies to an *item*, not the container. **align-self** uses the same values as **align-items**; it just works on one item at a time.

align-self

Values:	flex-start | flex-end | center | baseline | stretch
Default:	stretch
Applies to:	Flex items
Inherits:	No

In the following code and FIGURE 17-11, the fourth box is set to align at the end of the cross axis, while the others have the default stretch behavior:

```
.box4 {
  align-self: flex-end;
}
```

align-self: flex-end;

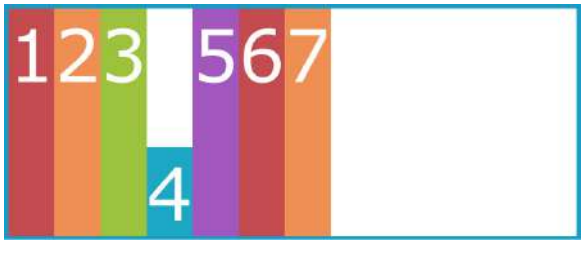


FIGURE 17-11. Use **align-self** to make one item override the cross-axis alignment of its container.

Aligning Multiple Lines

The final alignment option, **align-content**, affects how multiple flex lines are spread out across the cross axis. This property applies only when **flex-wrap** is set to **wrap** or **wrap-reverse** and there are multiple lines to align. If the items are on a single line, it does nothing.

align-content applies only when there are multiple wrapped flex lines.

align-content

Values:	flex-start | flex-end | center | space-around | space-between | stretch
Default:	stretch
Applies to:	Multiline flex containers
Inherits:	No

All of the values you see in the property listing should look familiar, and they work the way you would expect. This time, however, they apply to how extra space is distributed around multiple lines on the cross axis, as shown in FIGURE 17-12.

Again, the **align-content** property applies to the flex container element. A height is required for the container as well, because without it the container would be just tall enough to accommodate the content and there would be no space left over. For the alignment options shown in FIGURE 17-12, I gave the container a height of 350px and allowed the lines to wrap:

```
#container {
  display: flex;
  flex-direction: row;
  flex-wrap: wrap;
  height: 350px;
  align-content: flex-start;
}

.box {
  width: 25%;
}
```

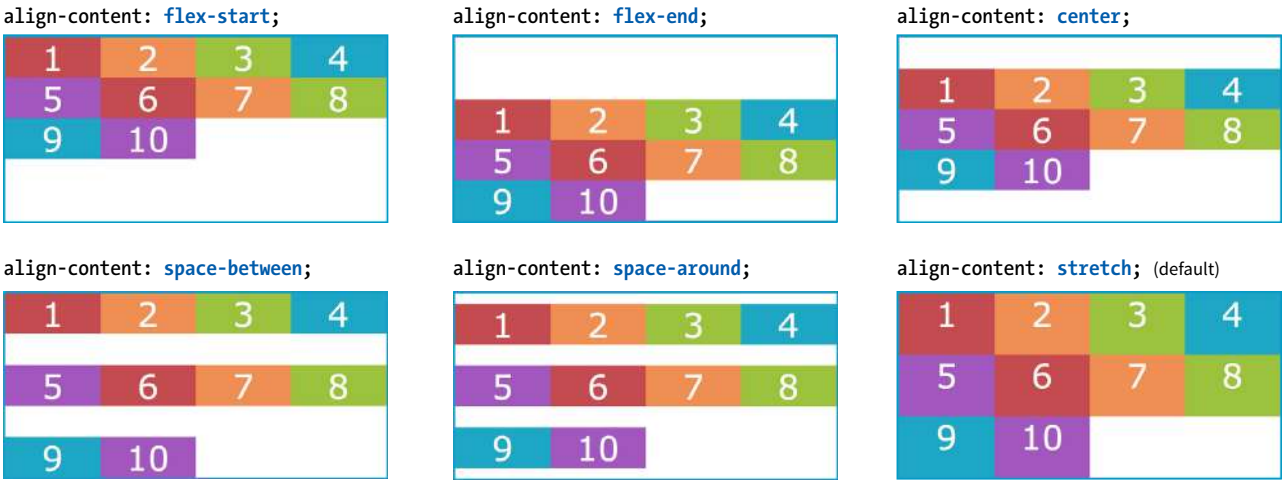


FIGURE 17-12. The **align-content** property distributes space around multiple flex lines.

FLEXBOX ALIGNMENT TECHNIQUES

We've only covered the flex container properties, and it turns out you already have enough Flexbox know-how to take on two popular methods for using Flexbox for alignment. In the first technique, we apply a margin to a flex item to set it apart from the other items on the axis. The second technique uses Flexbox to perfectly center an element within its containing element.

Aligning Items with Margins

Use margins to add space on the sides of particular flex items.

Menu bars are ubiquitous on the web, and it is common for one element in the bar, such as a logo or a search field, to be set off visually from the others. You can use a margin to put the extra container space on a specified side or sides of a flex item, thus setting that item apart. This should be clearer with an example.

The menu in FIGURE 17-13 has a logo and four menu options. I'd like the logo to stay in the left corner but the options to appear to the right, regardless of the width of the viewport.

THE MARKUP

```
<ul>
  <li class="logo"><img src="logo.png" alt="LoGoCo"></li>
  <li>About</li>
  <li>Blog</li>
  <li>Shop</li>
  <li>Contact</li>
</ul>
```

THE STYLES

```
ul {
  display: flex;
  align-items: center;
  background-color: #00af8f;
  list-style: none; /* removes bullets */
  padding: .5em;
  margin: 0;
}
li {
  margin: 0 1em;
}
li.logo {
  margin-right: auto;
}
```

FIGURE 17-13. Using a margin to adjust the space around flex items. In this example, the right margin of the logo item pushes the remaining items to the right.

I've turned the unordered list (`ul`) into a flex container, so its list items (`li`) are now flex items. By default, the items would stay together at the start of the main axis (on the left) with the extra space on the right. Setting the right margin on the logo item to `auto` moves the extra space to the right of the logo, pushing the remaining items all the way to the right (the "main end").

This technique applies to a number of scenarios. If you want just the last item to appear on the right, set its left margin to `auto`. Want equal space around the center item in a list? Set both its left and right margins to `auto`. Want to push a button to the bottom of a column? Set the top margin of the last item to `auto`. The extra space in the container goes into the margin and pushes the neighboring items away. Note, however, that when you apply `margin: auto` to a flex item, the `justify-content` property (if there is one) no longer has a visual effect, because you've manually assigned a location for the extra space on the main axis.

Centering Elements with Flexbox

You may be surprised to know that CSS didn't really have an explicit way to center items vertically until Flexbox came along. Sure, we came up with workarounds, but it was a relief to us veterans to have a simple and correct solution.

FIGURE 17-14. Centering an element horizontally and vertically using Flexbox alignment properties

In the following example, I've centered a `div` that contains a single image in a larger containing `div` using Flexbox alignment properties with the `center` keyword value (FIGURE 17-14). It really couldn't be simpler, and yet we waited years and years for this!

THE MARKUP

```
<div class="container">
  <div class="item"><img src="flower.png" alt="photo of a yellow
flower"></div>
</div>
```

THE STYLES

```
.container {
  display: flex;
  justify-content: center;
  align-items: center;
  width: 400px;
  height: 400px;
  background: blanchedalmond;
}
.item {
  width: 100px;
  height: 100px;
}
```

We've covered a lot of territory, so it's a good time to try out Flexbox alignment features in EXERCISE 17-2.

EXERCISE 17-2. A flexible online menu

Now it's time for you to play around with Flexbox alignment properties by using content a bit more complex than links in a menu bar. In this exercise, you'll format a simple online menu with a number of menu items. As always, the materials are available at *learningwebdesign.com/6e/materials*.

Open *flex-menu.html* in a text or code editor, and you'll see that it has all of the content ready to go as well as an internal style sheet with styles for the cosmetic aspects of the menu (colors, fonts, borders, spacing, etc.). Open the file in a browser, and the menu items should appear in a column because they are block elements. The border on the **#menu** wrapper **div** shows its boundaries.

1. First, we'll go for maximum impact with minimal effort by making the **#menu** wrapper a flex container. There is already a rule for **#menu**, so add this declaration to it:

   ```
   #menu {
     border: 3px solid #783F27;
     display: flex;
   }
   ```

   Save the file and reload the page in the browser, and BAM!… they're in a row! Because we haven't added any other flex properties, they are demonstrating default Flexbox behavior (FIGURE 17-15):

   - Each item (defined by a **section** element) is the full height of the **#menu** container, regardless of its content, because the default cross-axis alignment is **stretch**.
   - The sections have their widths set to 240 pixels, which is preserved by default. Depending on the width of your browser window, you may see content extending beyond the container and getting clipped off, as shown in the figure.

2. By default, flex items appear in the writing direction (a row, left to right, in English). Add the **flex-direction** property to the existing **#menu** rule to try out some of the other values (**row-reverse**, **column**, **column-reverse**). The items are numbered to make their order more apparent.

   ```
   flex-direction: row-reverse;
   ```

3. Set the **flex-direction** back to **row**, and let's play around with the cross-axis alignment by using the **align-items** property. Begin by setting it to **flex-start** (FIGURE 17-16). Save and reload, and see that the items all line up at the start of the cross axis (the top, in this case) and the height no longer stretches to fill the height of the container. In fact, the container height expands just enough to fit the longest item. Try some of the other values for **align-items** (**flex-end**, **center**, **baseline**, and **stretch**) to get a feel for how each behaves.

   ```
   align-items: flex-start;
   ```

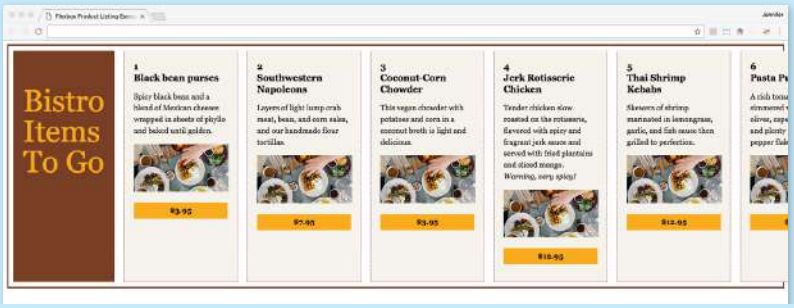


FIGURE 17-15. By default, the items stay in one row even though there is not enough room for them and content gets clipped.

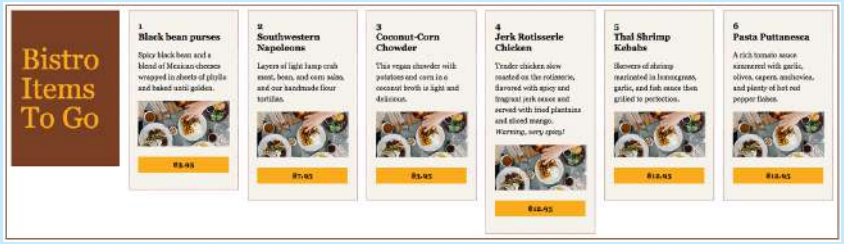


FIGURE 17-16. Using the **align-items** property to align the items at the start of the cross axis (**flex-start**)

EXERCISE 17–2. Continued

4. When you are done experimenting, set **align-items** back to **stretch**. Instead of having all the items on one line and getting cropped by the edge of the browser, let's have them wrap onto multiple lines by using the **flex-wrap** property on the **#menu** container:

   ```
   flex-wrap: wrap;
   ```

   Save the file and reload the page in your browser (FIGURE 17-17). Resize the browser window and watch the lines rewrap. Notice that each flex line is as tall as the tallest item in that row, but rows may have different heights based on item content.

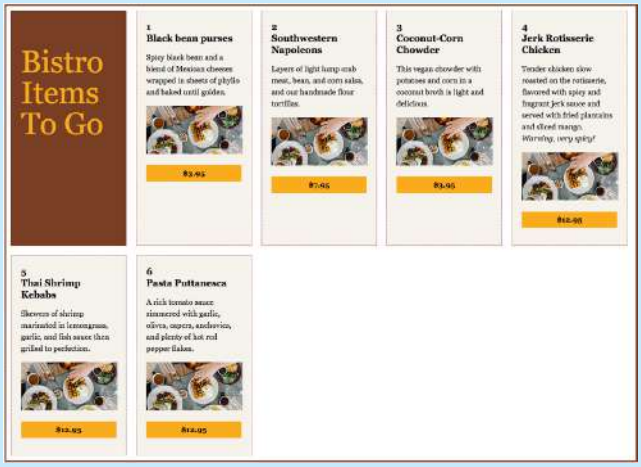


FIGURE 17-17. The menu with wrapping turned on

5. If you'd like, you can replace the **flex-direction** and **flex-wrap** declarations with a single **flex-flow** declaration, like so:

   ```
   flex-flow: row wrap;
   ```

6. By default, the items on each flex line are stacked toward the start of the main axis (the left). Try changing the main-axis alignment of items with the **justify-content** property (again, applied to the **#menu** flex container rule). They look good centered in the container, but check out the effect of the other values (**flex-start**, **flex-end**, **space-between**, **space-around**) as well.

   ```
   justify-content: center;
   ```

7. As a final tweak, let's make the price buttons line up at the bottom of each menu item, which is possible if each item is also a flex container. Make each item a nested flex container by setting its display property to **flex** and specifying the direction as **column** so they continue to stack up vertically:

   ```
   section {
     ...
     display: flex;
     flex-direction: column;
   }
   ```

   Now the **h2** and **p** elements are flex items within the **section** flex container. When you reload the page in the browser (don't forget to save your changes first!), it looks about the same as when the sections were made up of block elements. The subtle difference is that now the neighboring margins between elements stack up and do not collapse, because margins don't collapse on flex items.

   Now push the paragraphs containing the prices to the bottom using the **margin: auto** trick. Add this declaration to the existing style rule for elements with the class name "price":

   ```
   .price {
     ...
     margin-top: auto;
   }
   ```

FIGURE 17-18 shows the final state of the "Bistro Items To Go" menu with nested flexboxes. We'll continue working on this file after we've covered the item-specific properties, so save it for later.

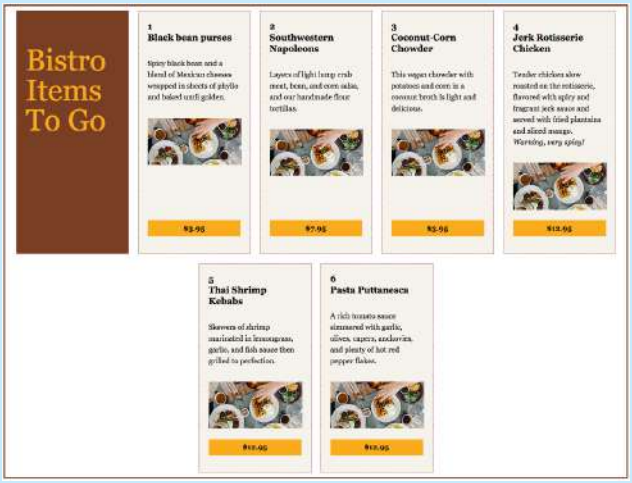


FIGURE 17-18. The menu so far with wrapping flex items and aligned prices

DETERMINING HOW ITEMS “FLEX” IN THE CONTAINER

At last, we’ve arrived at the important property that gives Flexbox its name! As mentioned at the beginning of the chapter, the primary feature of the Flexbox model is that items can *flex* (resize) to fill the available space in the container. In this section, you’ll get to know the flex properties.

Earlier, you learned about the **`justify-content`** property, which distributes extra space in the container between and around items along the main axis. The concept of **flex** is concerned with how space is distributed *within* items themselves, growing or shrinking them as required to fit the available space.

Flex is controlled with the **`flex`** property, which specifies how much an item can grow and shrink, and identifies its starting size. The full story is that **`flex`** is a shorthand property for **`flex-grow`**, **`flex-shrink`**, and **`flex-basis`**, but the spec strongly recommends that authors use the **`flex`** shorthand instead of individual properties in order to avoid conflicting default values and to ensure that authors consider all three aspects of **`flex`** for every instance.

`flex`

Values: none | *flex-grow flex-shrink flex-basis*

Default: `0 1 auto`

Applies to: Flex items

Inherits: No

The value for the **`flex`** property is typically values for the three flex properties, listed in this order:

```
flex: flex-grow flex-shrink flex-basis;
```

For the **`flex-grow`** and **`flex-shrink`** properties, the values 1 and 0 work like on/off switches, where 1 “turns on” the behavior, allowing an item to grow or shrink, and 0 prevents it. The **`flex-basis`** property sets the starting dimensions, either to a specific size or a size based on the content.

To show you how this works, here is a quick example in which a list item starts at 200 pixels wide (its **`flex-basis`**), is allowed to expand to fill extra space (**`flex-grow`** set to 1), but is not allowed to shrink (**`flex-shrink`** set to 0) narrower than the original 200 pixels:

```
li {
  flex: 1 0 200px;
}
```

That should give you the general idea. In this section, we’ll take a much closer look at growing, shrinking, and base size, in that order. The flex properties, although easy to use, are a bit tedious to explain, so take a deep breath as we head into the weeds.

The flex properties apply to flex items, not the container.

First, it is important to note that `flex` and its component properties apply to flex *items*, not the container. Keeping track of which properties go on the container and which go on items is one of the tricks of using Flexbox. See the **"Flex Properties"** sidebar for a handy list of how the properties are divided.

Expanding Items (flex-grow)

The first value in the `flex` property specifies whether (and in what proportion) an item may stretch larger—in other words, its `flex-grow` value (see **Note**). By default it is set to 0, which means an item is not permitted to grow wider than the size of its content or its specified width.

flex-grow

Values: *number*

Default: 0

Applies to: Flex items

Inherits: No

If you set the `flex-grow` value for all the items in a container to 1, the browser takes whatever extra space is available along the main axis and applies it equally to each item, allowing them all to stretch the same amount.

Let's take the simple box example from earlier in the chapter and see how it behaves with various flex settings applied. FIGURE 17-19 shows what happens when `flex-grow` is set to 1 for all box items (`flex-shrink` and `flex-basis` are left at their default values). Compare this to the same example with `flex-grow` set to the default 0 (this is the same behavior we observed back in FIGURE 17-2).

THE MARKUP

```
<div id="container">
  <div class="box box1">1</div>
  <div class="box box2">2</div>
  <div class="box box3">3</div>
  <div class="box box4">4</div>
  <div class="box box5">5</div>
</div>
```

THE STYLES

```
.box {
  ...
  flex: 1 1 auto;
}
```

If you assign a higher `flex-grow` integer to an item, it acts as a ratio that applies more space within that item. For example, giving "box4" the value `flex-grow: 3` means that it receives three times the amount of leftover space than the remaining items, set to `flex-grow: 1`. FIGURE 17-20 shows the result of this style rule:

■ AT A GLANCE

Flex Properties

Now that you've been introduced to all the properties in CSS Flexible Box Layout Module Level 1, it might be helpful to see at a glance which properties apply to containers and which are set on flex items.

Container properties

Apply these properties to the flex container:

```
display
flex-flow
  flex-direction
  flex-wrap
justify-content
align-items
align-content
```

Flex item properties

Apply these properties to flex items:

```
align-self
flex
  flex-grow
  flex-shrink
  flex-basis
order
```

NOTE

`flex-grow` is the individual property that specifies how an item may expand. Authors are encouraged to use the shorthand `flex` property instead.

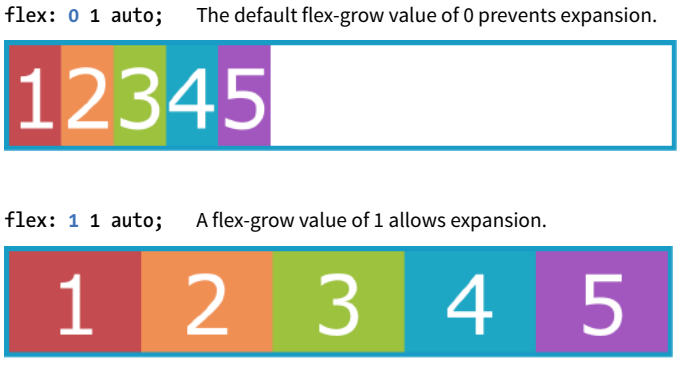


FIGURE 17-19. When `flex-grow` is set to 1, the extra space in the line is distributed into the items in equal portions, and they expand to fill the space at the same rate.

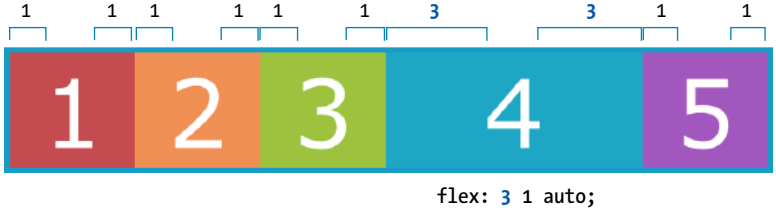


FIGURE 17-20. Assigning a different `flex-grow` value to an individual item. Here, "box4" was set to expand at three times the rate of the other items.

```
.box4 {
   flex: 3 1 auto;
}
```

Notice that the resulting item is not three times as wide as the others; it just got three times the amount of extra space added to it.

If there's not much space left over on the line, there's a chance that each portion of space could be small enough that it would not add up to much difference. You may need to play around with the **`flex-grow`** values and adjust the width of the browser until you get the effect you want.

Now that you have that concept down, shrinking should be straightforward, because it is based on the same principle.

Squishing Items (flex-shrink)

The second **`flex`** property value, **`flex-shrink`**, kicks in when the container is not wide enough to contain the items, resulting in a space deficit. It essentially takes away some space from within the items, shrinking them to fit, according to a specified ratio.

flex-shrink

Values: *number*

Default: 1

Applies to: Flex items

Inherits: No

NOTE

`flex-shrink` *is the individual property that specifies how an item may contract. Authors are encouraged to use the shorthand* `flex` *property instead.*

By default, the **`flex-shrink`** value is set to 1, which means if you do nothing, items shrink to fit at the same rate. When **`flex-shrink`** is 0, items are not permitted to shrink, and they may hang out of their container and extend beyond the edge of the viewport. Finally, as with **`flex-grow`**, a higher integer works as a ratio. An item with a **`flex-shrink`** of 2 will shrink twice as fast as if it were set to 1. You will not generally need to specify a shrink ratio value; just turning shrinking on (1) or off (0) should suffice.

By default, items may shrink when the container is not wide enough (flex-shrink: 1).

Eventually, we want our flex items to stop shrinking (this isn't *Honey, I Shrunk the Elements*). By default, this happens when they reach their minimum size, which is based on the smallest size the content can fit into (**`min-content`**). For example, the smallest possible width of an item containing the text "The dog jumps" is based on the word "jumps"—that's the smallest width the content can fit into without having to break any words into multiple lines.

You can easily adjust the minimum size by using the **`min-width`**/**`min-height`** properties to set the length to 0, 12em, or any other length that seems useful. Watch for this effect when deeply nested items force a flex item to be wider than expected.

You will see **`flex-shrink`** in action in FIGURE 17-21 in the next section.

Providing an Initial Size (flex-basis)

NOTE

`flex-basis` *is the individual property that sets the initial size of the item. Authors are encouraged to use the shorthand* `flex` *property instead.*

The third **`flex`** value defines the starting size of the item before any wrapping, growing, or shrinking occurs (**`flex-basis`**). It may be used instead of the **`width`** property (or **`height`** property for columns) for flex items.

flex-basis

Values: *length* | *percentage* | `content` | `auto`

Default: `auto`

Applies to: Flex items

Inherits: No

In this example, the **`flex-basis`** of the boxes is set to 100 pixels (FIGURE 17-21). The items are allowed to shrink smaller to fit in the available space (**`flex-shrink: 1`**), but they are not allowed to grow any wider (**`flex-grow: 0`**) than 100 pixels, leaving extra space in the container:

```
.box {
  flex: 0 1 100px;
}
```

`flex: 0 1 100px;`

When the container is wide, the items will not grow wider than their `flex-basis` of 100 pixels because `flex-grow` is set to 0.

When the container is narrow, the items are allowed to shrink to fit (`flex-shrink: 1`).

FIGURE 17-21. Using `flex-basis` to set the starting width for items

By default, **`flex-basis`** is set to **`auto`**, which uses the specified **`width`**/**`height`** property values for the item size. If the item's main size property (**`width`** or **`height`**) is not set, or is set to **`auto`** (its default), **`flex-basis`** uses the content width. You can also explicitly set **`flex-basis`** to be the width of the content with the **`content`** keyword.

Flex settings override specified widths/heights for flex items.

In this example, the **`flex-basis`** for the boxes would be 300 pixels because the **`auto`** value for **`flex-basis`** uses the value set by the **`width`** property. Items are allowed to grow wider, taking up any extra space in the container, but they are not allowed to shrink:

```
.box {
  width: 300px;
  flex: 1 0 auto;
}
```

When the browser goes about sizing a flex item, it consults the **`flex-basis`** value, compares it to the available space along the axis, and then adds or removes space from the item according to its grow and shrink settings. It's important to note that if you allow an item to grow or shrink, it could end up being narrower or wider than the length provided by **`flex-basis`** or **`width`**.

Handy Shortcut flex Values

The advantage to using the **`flex`** property is that there are some handy shortcut values that cover typical Flexbox scenarios. Some of the shortcut values override the defaults of the individual properties, but in the end it results in more predictable behaviors. When specifying **`flex`** for items, see if one of these easy settings will do the trick:

When specifying flex for items, see if you can take advantage of one of the handy flex shortcuts.

`flex: initial;`

This is the same as the default `flex: 0 1 auto`. It prevents the flex item from growing even when there is extra space, but allows it to shrink to fit in the container. The size is based on the specified `width`/`height` properties, defaulting to the size of the content. With the `initial` value, you can use `justify-content` for horizontal alignment.

`flex: auto;`

This is the same as `flex: 1 1 auto`. It allows items to be fully flexible, growing or shrinking as needed. The size is based on the specified `width`/`height` properties.

`flex: none;`

This is equivalent to `flex: 0 0 auto`. It creates a completely inflexible flex item, sizing it to the `width` and `height` property values. You can also use `justify-content` for alignment when `flex` is set to `none`.

`flex: integer;`

This is the same as `flex: integer 1 0%`. The result is a flexible item with a `flex-basis` of 0, which means it has **absolute flex** (see the sidebar **"Absolute Versus Relative Flex"**) and extra space is allocated using the specified integer as a ratio.

How are you doing? Are you hanging in there with all this Flexbox stuff? I know it's a lot to take in at once. We have just one more flex item property to cover before you get another chance to try it out yourself.

When to Reorder (and When Not To)

Keep in mind that although convenient, reordering is only a visual display sleight of hand and should be used with discretion. Some points to remember:

- Even though elements display in a different order for visual browsers, alternative devices such as screen readers still read the content in the order in which it appears in the source (although this behavior is not 100% reliable in the real world).
- Reorder the source if there is a logical (rather than visual) reason for the reordering.
- Don't use `order` because it is more convenient.
- Use `order` if the logical and visual order are *intended* to be disjointed.

CHANGING THE ORDER OF FLEX ITEMS

One useful feature of Flexbox is the ability to display items in an order that differs from their order in the source (see the sidebar **"When to Reorder (and When Not To)"**). That means you can change the layout order of elements by using CSS alone. This is a powerful tool for responsive design, allowing content from later in a document to be moved up on smaller screens.

To change the order of items, apply the `order` property to the particular item(s) you wish to move.

`order`

Values:	*integer*
Default:	0
Applies to:	Flex items and absolutely positioned children of flex containers
Inherits:	No

The value of the `order` property is a positive or negative number that affects the item's placement along the flex line. It is similar to the `z-index` property in that the specific number value doesn't matter, only how it relates to other values.

Absolute Versus Relative Flex

In FIGURE 17-20, we saw how extra space is assigned to items based on their flex ratios. This is called **relative flex**, and it is how extra space is handled whenever the `flex-basis` is set to any size other than zero (0), such as a particular `width`/`height` value or `auto`.

However, if you set the value of `flex-basis` to 0, something interesting happens. With a basis of 0, items get sized proportionally according to their flex ratios, which is known as **absolute flex**. So, with `flex-basis: 0%`, an item with a `flex-grow` value of 2 *would* be twice as wide as any items with a `flex-grow` value of 1. Again, this kicks in only when the `flex-basis` is 0.

In practice it is recommended that you always include a unit after the 0, such as `0px` or the preferred `0%`.

In this example of absolute flex, the first box is given a `flex-grow` value of 2, and the fourth box has a `flex-grow` value of 3 via the aforementioned shortcut `flex:` *`integer`*. In FIGURE 17-22, you can see that the resulting overall sizes of the boxes are in proportion to the `flex-grow` values because the `flex-basis` is set to 0:

```
.box {
  /* applied to all boxes */
  flex: 1 0 0%;
}
.box1 {
  flex: 2; /* shortcut value for flex: 2 1 0% */
}
.box4 {
  flex: 3; /* shortcut value for flex: 3 1 0% */
}
```

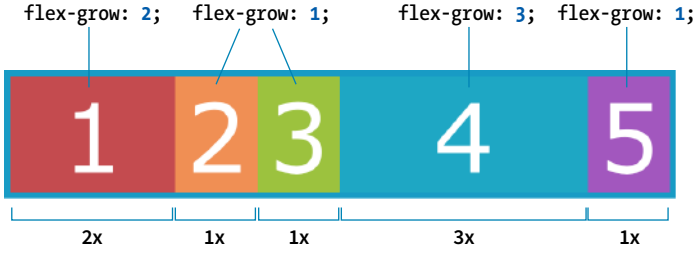


FIGURE 17-22. With absolute flex (when `flex-basis` is set to 0), boxes are sized proportionally according to their flex ratios.

By default, all items have an `order` value of zero (0). When items have the same `order` value, they are laid out in the order in which they appear in the HTML source. If they have *different* `order` values, they are arranged from the lowest `order` value to the highest.

Going back to our colorful numbered box example, here I've given `box3` an `order` value of 1. With a higher `order` value, it appears after all the items set to 0 (the default), as shown in FIGURE 17-23:

```
.box3 {
  order: 1;
}
```

FIGURE 17-23. Changing the order of items with the `order` property. Setting `box3` to `order: 1` makes it display after the rest.

When multiple items share the same **`order`** value, that group of value-sharing items (called an **ordinal group**) stick together and display in source order. What happens if we give **`box2`** an **`order`** value of 1 as well? Now both **`box2`** and **`box3`** have an **`order`** value of 1 (making them an ordinal group), and they get displayed in source order after all the items with the lower **`order`** value of 0 (FIGURE 17-24):

```
.box2, .box3 {
  order: 1;
}
```

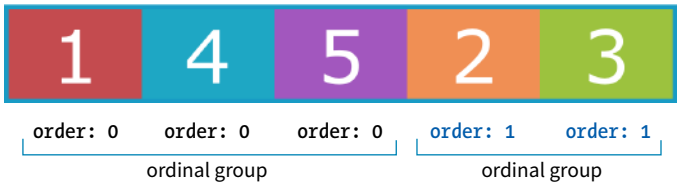


FIGURE 17-24. Setting `box2` to `order: 1` as well makes it display after the items with the default `order` of 0, but before `box3`.

You can also use negative values for **`order`**. To continue with our example, here I've given **`box5`** an **`order`** value of –1. Notice in FIGURE 17-25 that it doesn't just move back one space; it moves before all of the items that still have **`order`** set to 0, which is a higher value than –1:

```
.box5 {
  order: -1;
}
```

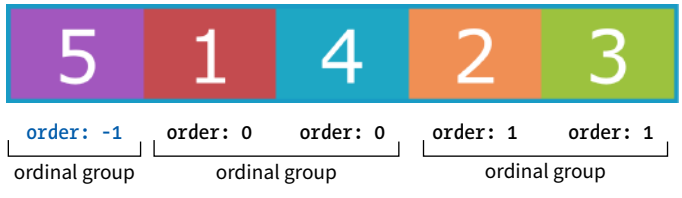


FIGURE 17-25. Negative values display before items with the default `order` of 0.

I've used simple values of 1 and –1 in these examples, but the result would be the same if I'd used values like 10008 or –649. The order goes from the lowest value to the highest, and the number values don't need to be in sequential order.

FLEXBOX FOR PAGE LAYOUT

Now let's take a look at how we can use **order** for something more useful than moving little boxes around in a line. Here is a simple document with a header, a main section consisting of an **article** and two **aside** elements, and a **footer** (see **Note**):

```
<header>...</header>
<main>
  <article><h2>Where It's At</h2></article>
  <aside id="news"><h2>News></h2></aside>
  <aside id="contact"><h2>Contact</h2><aside>
</main>
<footer>...<footer>
```

Here I've made the **main** element a flex container so the **article** and **aside** elements line up in a row, creating three columns (FIGURE 17-26). I set the **flex** factor for each item, allowing them to grow and shrink, and set their widths with **flex-basis**. Finally, I used the **order** property to specify the order in which I'd like them to appear. Notice that the Contact section is now first in the row, although it appears last in the source order. And, as an added bonus, all of the columns fill the height of the main container despite the amount of content in them because **stretch** is the default value for **align-items**:

```
main {
  display: flex;
}
article {
  flex: 1 1 50%;
  order: 2;
}
#news {
  flex: 1 1 25%;
  order: 3;
}
#contact {
  flex: 1 1 25%;
  order: 1;
}
```

Hip & Happenin' Headline
Contact
Where It's At
Integer ornare neque enim, non imperdiet ex pellentesque sed. Sed congue, nunc ut rhoncus consectetur, diam purus venenatis lacus, vel sollicitudin nibh tortor sed nunc. Vestibulum at ante ac tortor ultricies gravida eget molestie nibh. Fusce tempor dictum lectus, id luctus sapien tristique suscipit. Cras volutpat lectus at urna vulputate pellentesque. Praesent eleifend, sapien ut finibus facilisis, orci augue elementum leo, a faucibus augue nibh ac urna. Aliquam erat volutpat. Sed ultrices tempus neque, nec iaculis orci sollicitudin id. Mauris gravida congue sapien, vitae finibus nisi condimentum id. Donec turpis metus, euismod sed luctus sit amet, semper eu purus.
News
The small print

FIGURE 17-26. A columned layout using Flexbox

NOTE

Although it is possible to create a full-page layout with Flexbox, the task is more appropriately handled with Grid Layout, which I'll cover in **Chapter 18**. *However, because Flexbox has slightly better browser support than Grid Layout, it may be a suitable fallback. Flexbox is better suited for designing individual components on the page such as navigation bars, a series of product "cards," or anything that you want to put in a line.*

That concludes our tour of Flexbox properties! In EXERCISE 17-3, you can put some of the item-level properties to use in the Bistro menu, including changing the order in which menu item components appear.

MORE FLEXBOX RESOURCES

You've learned the ins and outs of Flexbox in this chapter, but it never hurts to get additional perspectives and try some hands-on tutorials. Following are some of the sites that I've found useful or entertaining. Note that if you do your own web search, you should limit your findings to posts from 2016 and later, or you may come across outdated advice based on earlier spec versions.

CSS Flexible Box Layout Module Level 1 *(w3.org/TR/css-flexbox/)*

The definitive source for Flexbox properties is the Flexbox spec itself.

CSS Flexbox Layout Guide *(css-tricks.com/a-guide-to-flexbox/)*

This summary of Flexbox features by Chris Coyier is one of the most popular Flexbox references out there. Some developers keep it open in a browser when they do Flexbox work.

EXERCISE 17-3. Adjusting flex and order

The online menu is looking pretty good, but let's put a few finishing touches on it. Open the *flex-menu.html* file as you left it at the end of EXERCISE 17-2.

1. Instead of having lots of empty space inside the menu container, we'll make the items fill the available space. Because we want the items to be fully flexible, we can use the `auto` value for `flex` (the same as `flex: 1 1 auto`). Add this declaration to the `section` rule to turn on the stretching behavior:

```
section {
  ...
  flex: auto;
}
```

2. OK, one last tweak: let's make the photos appear at the top of each menu item. Because each section is a flex container, we can use the `order` property to move its items around. In this case, create a new rule that selects the paragraphs with the "photo" `class` name and give it a value less than the default 0. This will make the photos display at the top of each listing (FIGURE 17-27):

```
.photo {
  order: -1;
}
```

It's not the best-looking web page in the world, but you got a chance to try out a lot of the Flexbox properties along the way!

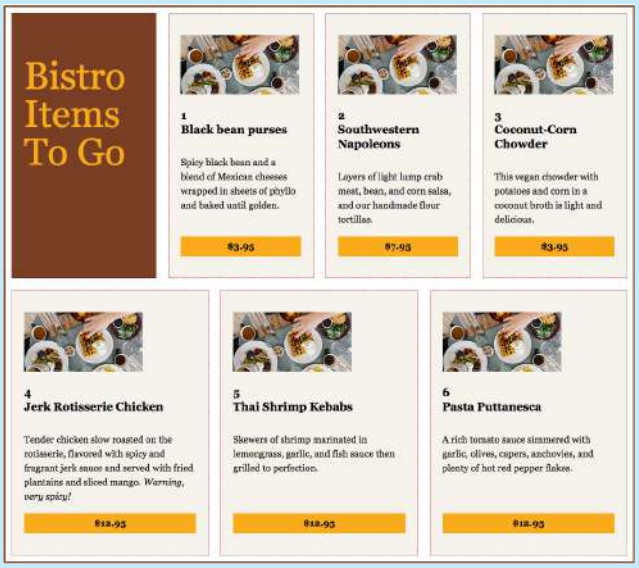

Bistro Items To Go
1
Black bean purses
Spicy black bean and a blend of Mexican cheeses wrapped in sheets of phyllo and baked until golden.
$3.95
2
Southwestern Napoleons
Layers of light lump crab meat, bean, and corn salsa, and our handmade flour tortillas.
$7.95
3
Coconut-Corn Chowder
This vegan chowder with potatoes and corn in a coconut broth is light and delicious.
$3.95
4
Jerk Rotisserie Chicken
Tender chicken slow roasted on the rotisserie, flavored with spicy and fragrant jerk sauce and served with fried plantains and sliced mango. Warning, very spicy!
$12.95
5
Thai Shrimp Kebabs
Skewers of shrimp marinated in lemongrass, garlic, and fish sauce then grilled to perfection.
$12.95
6
Pasta Puttanesca
A rich tomato sauce simmered with garlic, olives, capers, anchovies, and plenty of hot red pepper flakes.
$12.95

FIGURE 17-27. The final Bistro menu with items flexing to fill the extra space and the photos moved to the top of each listing

MDN Web Docs Flexbox Guides (*oreil.ly/WHVhe*)

Once again, MDN Web Docs is a great source for thorough and easy-to-understand overviews of Flexbox features. See also the "CSS Flexible Box Layout" reference at *oreil.ly/a5Qx8t*.

Flexbox Froggy (*flexboxfroggy.com*)

Don't miss this online game for learning Flexbox by helping colorful frogs make it back to their lily pads.

What the Flexbox?! (*flexbox.io*)

Wes Bos does a great job of walking you through Flexbox properties as well as a few code projects in this free, 20-part video series.

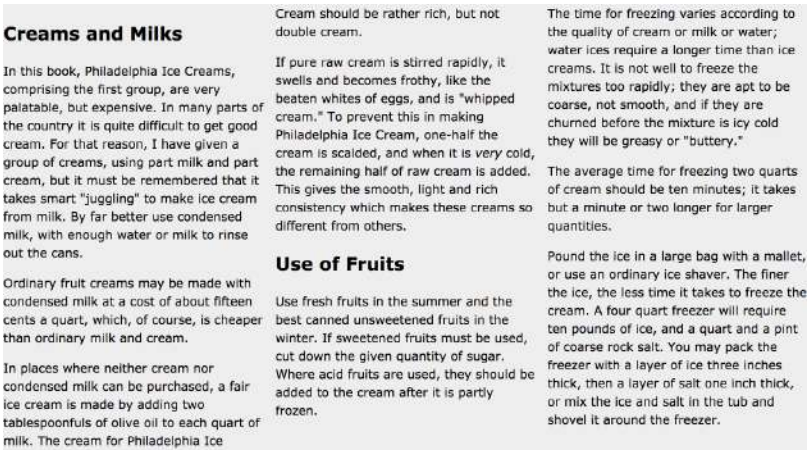
Flexbox Playground (*oreil.ly/YhPJO*)

As the name says, this page by Gabi lets you play around with all of the Flexbox properties and values and see the results instantly. It's a nice way to get familiar with what Flexbox can do.

Multi-Column Layout

Another CSS layout tool you may want to try is Multi-Column Layout. CSS Multi-Column Layout Module Level 1 (*w3.org/TR/css-multicol-1*) provides tools for pouring text content into a number of columns, as you might see in a newspaper (FIGURE 17-28). It is designed to be flexible, allowing the widths and number of columns to fit the available space automatically.

You can read more about this technique in my article "Multicolumn Layout," available at *learningwebdesign.com/articles/*.



Creams and Milks

In this book, Philadelphia Ice Creams, comprising the first group, are very palatable, but expensive. In many parts of the country it is quite difficult to get good cream. For that reason, I have given a group of creams, using part milk and part cream, but it must be remembered that it takes smart "juggling" to make ice cream from milk. By far better use condensed milk, with enough water or milk to rinse out the cans.

Ordinary fruit creams may be made with condensed milk at a cost of about fifteen cents a quart, which, of course, is cheaper than ordinary milk and cream.

In places where neither cream nor condensed milk can be purchased, a fair ice cream is made by adding two tablespoonfuls of olive oil to each quart of milk. The cream for Philadelphia Ice Cream should be rather rich, but not double cream.

If pure raw cream is stirred rapidly, it swells and becomes frothy, like the beaten whites of eggs, and is "whipped cream." To prevent this in making Philadelphia Ice Cream, one-half the cream is scalded, and when it is *very* cold, the remaining half of raw cream is added. This gives the smooth, light and rich consistency which makes these creams so different from others.

Use of Fruits

Use fresh fruits in the summer and the best canned unsweetened fruits in the winter. If sweetened fruits must be used, cut down the given quantity of sugar. Where acid fruits are used, they should be added to the cream after it is partly frozen.

The time for freezing varies according to the quality of cream or milk or water; water ices require a longer time than ice creams. It is not well to freeze the mixtures too rapidly; they are apt to be coarse, not smooth, and if they are churned before the mixture is icy cold they will be greasy or "buttery."

The average time for freezing two quarts of cream should be ten minutes; it takes but a minute or two longer for larger quantities.

Pound the ice in a large bag with a mallet, or use an ordinary ice shaver. The finer the ice, the less time it takes to freeze the cream. A four quart freezer will require ten pounds of ice, and a quart and a pint of coarse rock salt. You may pack the freezer with a layer of ice three inches thick, then a layer of salt one inch thick, or mix the ice and salt in the tub and shovel it around the freezer.

FIGURE 17-28. An example of text formatted with the multicolumn properties

TEST YOURSELF

We covered lots of ground in this chapter. See how you do on this quiz covering some of the highlights. As always, **Appendix A** has the answers.

1. How do you turn an element into a flex item?

2. Match the properties with their functions.

a. ____	`justify-content`	1. Distribute space around and between flex lines on the cross axis.
b. ____	`align-self`	2. Distribute space around and between items on the main axis.
c. ____	`align-content`	3. Position items on the cross axis.
d. ____	`align-items`	4. Position a particular item on the cross axis.

3. How is **`align-items`** different from **`align-content`**? What do they have in common?

4. Match the properties and values to the resulting effects.

a. ____	`flex: 0 1 auto;`	1. Items are completely inflexible, neither shrinking nor growing.
b. ____	`flex: none;`	2. Item will be twice as wide as others with **`flex: 1`** and may also shrink.
c. ____	`flex: 1 1 auto;`	3. Items are fully flexible.
d. ____	`flex: 2 1 0px;`	4. Items can shrink but not grow bigger.

5. Match the following **`flex-flow`** declarations with the resulting flexboxes (FIGURE 17-29).

a. `flex-flow: row wrap;` ____

b. `flex-flow: column nowrap;` ____

c. `flex-flow: row wrap-reverse;` ____

d. `flex-flow: column wrap-reverse;` ____

e. `flex-flow: row nowrap;` ____

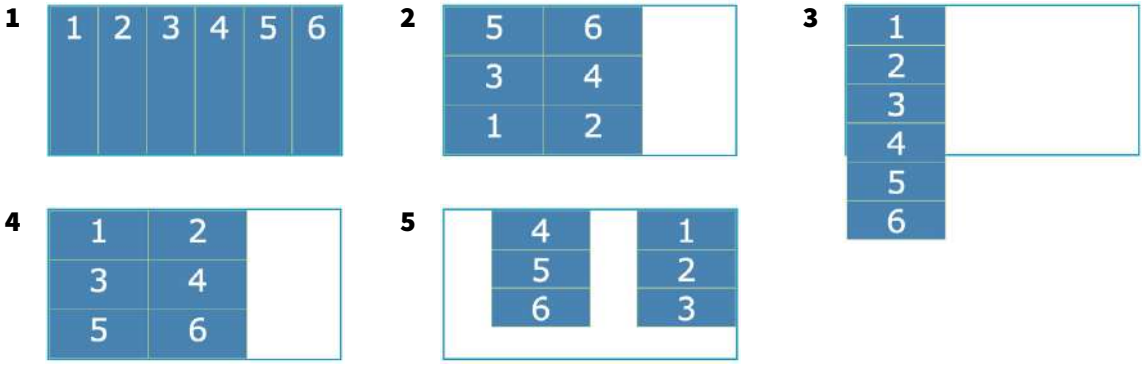


FIGURE 17-29. Various `flex-flow` settings

6. Write style rules for displaying the flex items in the order shown in FIGURE 17-30.

Source HTML

```
<div id="container">
 <div class="box box1">1</div>
 <div class="box box2">2</div>
 <div class="box box3">3</div>
 <div class="box box4">4</div>
 <div class="box box5">5</div>
 <div class="box box6">6</div>
 <div class="box box7">7</div>
</div>
```

After reordering

FIGURE 17-30. Write style rules to put the items in the shown order.

CSS REVIEW: FLEXBOX PROPERTIES

Here are the properties covered in this chapter, sorted according to whether they apply to the container or items.

Flex container properties

`display: flex`	Turns on flexbox mode and makes the element a flex container
`flex-direction`	Indicates the direction in which items are placed in the flex container
`flex-wrap`	Specifies whether the flex items are forced onto a single line or wrapped onto multiple lines
`flex-flow`	Shorthand property for **`flex-direction`** and **`flex-wrap`**
`justify-content`	Specifies how space is distributed between and around items on the main axis

`align-content`	Aligns flex lines within the flex container when there is extra space on the cross axis
`align-items`	Specifies how space is distributed around items on the cross axis

Flex item properties

`flex-grow`	Specifies how much a flex item is permitted to grow when there is extra space in the container
`flex-shrink`	Specifies how much a flex item is permitted to shrink when there is not enough room in the container
`flex-basis`	Indicates the initial main size of a flex item
`flex`	Shorthand property for **`flex-grow`**, **`flex-shrink`**, and **`flex-basis`**; specifies how items alter their dimensions to fit the available space
`align-self`	Specifies how one item is aligned on the cross axis (overrides **`align-items`**)
`order`	Indicates the order used to lay out items in their container

CHAPTER 18

GRID LAYOUT

IN THIS CHAPTER

How CSS Grid Layout works

Working with implicit grids

Placing items in the grid

Implicit grid features

Grid item alignment

We humans (well, a lot of us anyway) like to line things up in rows and columns, so it's surprising that it took the W3C and browsers over a quarter of a century to provide a reliable way to lay out elements and pages in a grid. Sure, we managed fancy layouts with tables and floats, but those were essentially hacks. The CSS Grid Layout Module provides a true system for laying out elements in rows *and* columns (remember that Flexbox lays out elements on one axis only). Grids can remain completely flexible to fit a variety of screen sizes, or they can be rigid, to mimic a printed page layout. You can use them for page components that organize a number of individual items (like a photo gallery) or for creating sophisticated, whole-page layouts such as those designed by Jen Simmons in her Lab demos (FIGURE 18-1).

CSS Grid Layout Module Level 2 (*w3.org/TR/css-grid-2/*) is one of the more complex modules in CSS, so we have a lot of ground to cover in this chapter. Here's the plan:

A simple photo gallery

Jen Simmons' sophisticated grid layout experiments from *labs.jensimmons.com*

FIGURE 18-1. Examples of Grid Layout in use

- We'll start with a bird's-eye view of how CSS Grid Layout works.
- We'll look at some default behaviors of Grid Layout with a lesson on "implicit" grids.
- From there, we'll set up "explicit" grids with specific numbers of rows and columns.
- We'll review the many options for placing grid items into the grid.
- Next, you'll learn about grid alignment—both the alignment of rows and columns within the grid and the alignment of items within their grid areas.
- Finally, I'll briefly introduce some techniques for making grids work in responsive layouts.

Using Grid Layout isn't actually that complicated. What makes it *look* complicated is that the spec provides *so* many options for specifying every little thing. All those options are terrific for customizing production work, but they can feel cumbersome when you're learning to use it. We'll take things one step at a time, and by the end of the chapter you'll have a solid CSS Grid Layout toolbox. Ready to dive in?

BROWSER SUPPORT NOTE

CSS Grid Layout has been supported by all modern browsers since 2017, and as of this writing, it works as described here for 98% of browsers worldwide (according to CanIUse.com). The version of Grid Layout described here does not work on any version of Internet Explorer, so you may want to provide an alternative layout as a backup if you know your specific audience is using that browser. CSS feature queries that check for Grid Layout support are a good way to get your grid-based layouts to the browsers that can handle them and an alternative layout to the other 2%. Feature queries are discussed in detail in **Chapter 21, More CSS Techniques**.

GRID LAYOUT OVERVIEW

Let's begin by getting familiar with how grids work. **Grid Layout** is a display mode that allows items to be organized in two dimensions (rows and columns). You just learned about the Flexbox display mode, which, by contrast, organizes elements along one axis only. FIGURE 18-2 compares the same gallery set with `display: flex` versus `display: grid`.

`display: flex;`
Items line up horizontally

`display: grid;`
Items line up horizontally and vertically

FIGURE 18-2. The difference between Flexbox and Grid Layout

You can see that in the Grid Layout example, all of the images line up neatly under one another along their left edges. In the Flexbox example, although the images flow onto multiple lines, the images in the second and third lines do not line up with the images on the lines above. Keep in mind that if you

have content that doesn't need to be aligned in columns, Flexbox may be the only tool you need. In practice, it is common for Grid Layout and Flexbox to be used together to achieve the layout of a page and its components.

Grid Terminology

Before we get to the code, let's pause here to look at the parts of a grid according to the spec. A grid has a number of components, as illustrated in FIGURE 18-3.

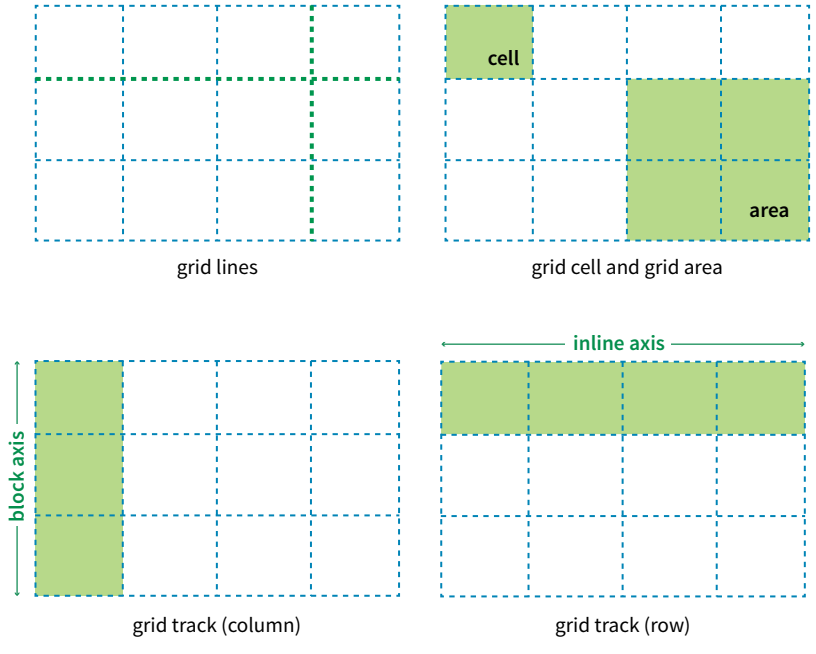


FIGURE 18-3. The parts of a CSS grid

These terms will be useful for upcoming discussions:

grid line

The horizontal and vertical dividing lines of the grid are called **grid lines**.

grid cell

The smallest unit of a grid is a **grid cell**, which is bordered by four adjacent grid lines (with no grid lines running through it).

grid area

A **grid area** is an area made up of one or more adjacent grid cells. A grid area must be rectangular; in other words, it may not be shaped like an "L" or any other shape.

grid track

The space between two adjacent grid lines is a **grid track**, which is a generic name for a **grid row** or a **grid column**.

inline axis

The **inline axis** is the direction in which inline text flows based on the language of the document. In English, text flows horizontally; therefore, the inline axis in grids is horizontal. Grid rows follow the inline axis.

block axis

The **block axis** is the direction that blocks appear in a block layout, based on language direction. In English, the block axis is vertical because blocks are laid out from top to bottom. Grid columns follow the block axis.

Grid Containers and Items

A grid is made up of a container and some number of items. When you apply the `display: grid` property to an element, it becomes the **grid container** and defines the context for grid formatting. All of its direct child elements automatically become **grid items** that end up positioned in the grid. If you've just read **Chapter 17, Getting in Line with Flexbox**, this children-become-items scheme should sound familiar.

The important words in the previous paragraph are "direct child," as only those elements become grid items. Elements contained within those elements do not take part in the grid, so you can't place nested elements. You can, however, nest a grid inside another grid if you need to apply a grid to a deeper level.

To turn an element into a grid container, set its `display` property to `grid` or `inline-grid` (see **Note**).

NOTE

Inline grids function the same as block-level grids, but they can be used in the flow of content. I'll focus only on block-level grids in this chapter.

In this simple example, the `#layout div` becomes a grid container, and each of its children (`#one`, `#two`, `#three`, `#four`, and `#five`) becomes a grid item.

THE MARKUP

```
<div id="layout">
  <div id="one">One</div>
  <div id="two">Two</div>
  <div id="three">Three</div>
  <div id="four">Four</div>
  <div id="five">Five</div>
</div>
```

THE STYLES

```
#layout {
  display: grid;
}
```

Now that you know a few basics, let's look at what the grid algorithm can do right out of the box with very little instruction.

IMPLICIT GRIDS AND DEFAULT GRID BEHAVIOR

FIGURE 18-4 shows what the previous example with five grid items looks like if you open it in a browser. I've applied some font styles and background colors that are not shown in the example code to make the structure clear.

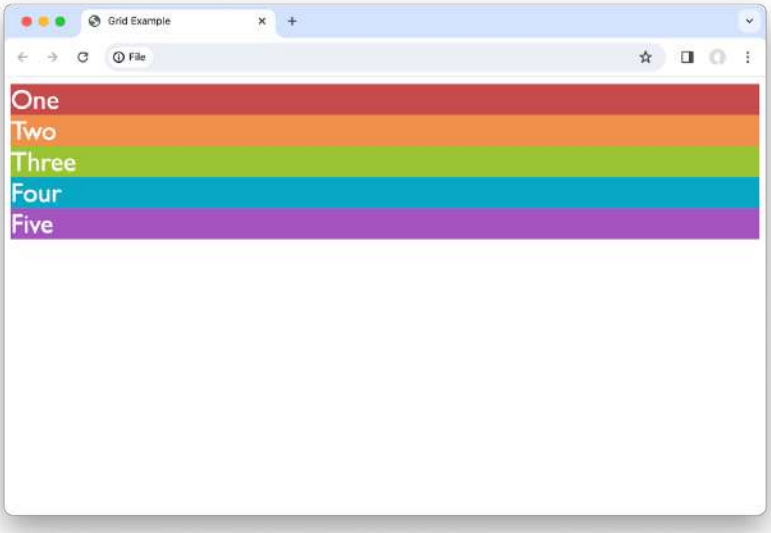


FIGURE 18-4. A minimal, default grid with five items

Without specifying any additional grid-related styles, we end up with a grid with one column (the default) and five rows. Even though it looks like nothing much is happening, there are some important grid behaviors going on here:

- One grid item is placed in each cell, which is the default behavior.
- The grid algorithm automatically generates enough rows to place all of the grid items in the container. In this example, we have five rows for five items because our grid has only one column (the default).
- By default, the height of the rows is just tall enough to contain the item's content (`auto`).

A grid that results from the browser automatically adding rows or columns to contain all the items is called an **implicit grid**. It's what you get when you just let the grid algorithm do its thing. By contrast, when you define a grid with a specific number of rows *and* columns, you create an **explicit grid**. For now, we're going to focus on implicit grid behavior.

AT A GLANCE

Grid Properties

Here's a handy list of the grid properties, organized by whether they apply to the container or to individual grid items.

Grid container properties

```
display: grid | inline-grid
grid
  grid-template
    grid-template-rows
    grid-template-columns
    grid-template-areas
  grid-auto-rows
  grid-auto-columns
  grid-auto-flow
gap
  row-gap
  column-gap
justify-items
align-items
justify-content
align-content
```

Grid item properties

```
grid-column
  grid-column-start
  grid-column-end
grid-row
  grid-row-start
  grid-row-end
grid-area
justify-self
align-self
order (not in Grid Layout module)
z-index (not in Grid Layout module)
```

Let's give that same grid three columns, each 300 pixels wide, with the **grid-template-columns** property. I'll be talking a lot more about this property and its values later in the chapter, but for now just pay attention to how the items are automatically placed in the grid cells (FIGURE 18-5):

```
#layout {
  display: grid;
  grid-template-columns: 300px 300px 300px;
}
```

A grid that results from the browser automatically adding rows or columns to contain all the items is called an implicit grid.

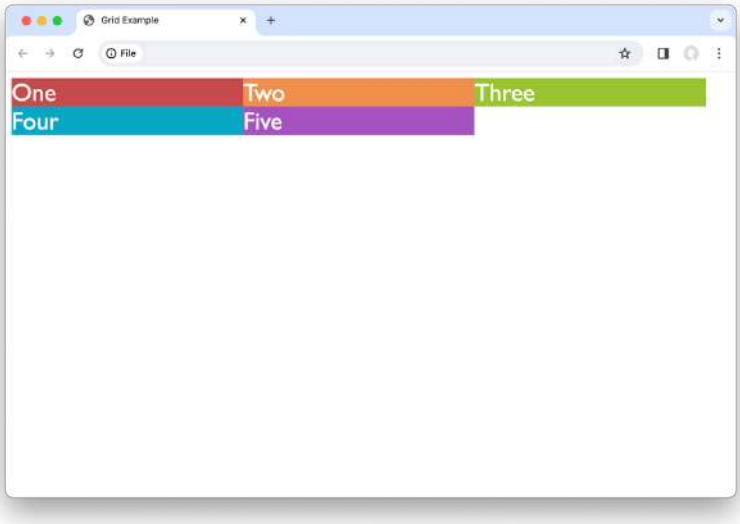


FIGURE 18-5. The example grid with three columns

Notice that the items are placed from left to right, and a new row is added below the first. That's because grids are aware of the writing direction of the document they are in. The example document is written in English, so items automatically flow into rows from left to right and from top to bottom (see **Note**). In an Arabic or Hebrew document, rows would fill from the right edge to the left.

NOTE

Because this book is written in English, I'll focus on the default left to right (inline), top to bottom (block) grid orientation. If you are creating a site in Arabic, Hebrew, or a vertically oriented language, know that your grids will default to the direction that feels familiar.

The automatic row- and column-generating behavior of the grid algorithm is really handy, and for some purposes, an implicit grid may be all you need. For example, if you just want an orderly presentation of photos in a gallery (like the one in FIGURE 18-2), you can specify the number of columns and let the browser handle the rest. The implicit grid is also useful when you have an unknown number of items or if the number of items in the container may change dynamically.

Implicit Grid Properties

The Grid Layout module provides a few properties for controlling the size of automatically generated tracks and the manner in which items flow into them.

Implicit track sizing

As you've seen in the previous examples, any row or column added to an implicit grid will be sized just large enough to accommodate the height or width of the contents. If you want to give automatically generated rows and columns specific dimensions, use the **`grid-auto-`** properties.

`grid-auto-rows`
`grid-auto-columns`

Values: *list of track sizes*
Default: auto
Applies to: Grid containers
Inherits: No

The **`grid-auto-row`** and **`grid-auto-columns`** properties apply to the grid container. Their values are a list of one or more track sizes to be used for automatically generated tracks (thus the term "auto" in the property names). To make the rows of our sample implicit grid 4em high (FIGURE 18-6), add this declaration to the styles for the grid container:

```
#layout {
  display: grid;
  grid-template-columns: 300px 300px 300px;
  grid-auto-rows: 4em;
}
```

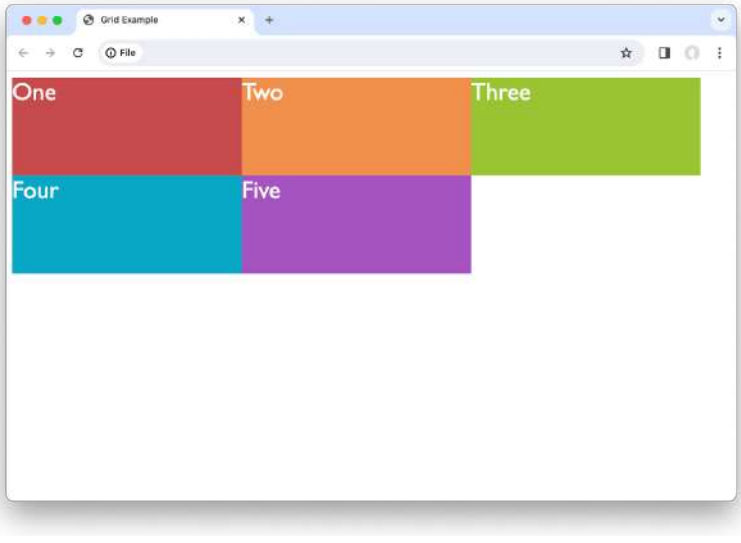


FIGURE 18-6. Specify a height for automatically generated rows with **`grid-auto-rows`**

Flow direction

grid-auto-flow

Values: `row` or `column` | `dense` *(optional)*

Default: `row`

Applies to: Grid containers

Inherits: No

So far, we've only looked at items flowing into rows, but you can use the **`grid-auto-flow`** property to make items flow in by columns instead, as shown in this example:

```
#listings {
  display: grid;
  grid-auto-flow: column;
}
```

If the height of the grid container is limited, the grid algorithm automatically adds implicit columns to the right until all the grid items are accommodated, just as we saw for rows. You'll see this in action in EXERCISE 18-1.

Grid density

By default, items are placed in the first area in which they fit. Cells that are too small to accommodate the content will be skipped over until a cell large enough to place it in is found. If you include the optional **`dense`** keyword for the **`grid-auto-flow`** property, it instructs the browser to fill the grid as densely as possible, allowing the items to appear out of sequence in order to fill the available space:

```
#listings {
  display: grid;
  grid-auto-flow: dense rows;
}
```

The example on the left in FIGURE 18-7 shows the default flow method. I've used numbered, colored blocks, but you might imagine these are photos with intrinsic dimensions. Look closely, and you'll see that the grid items are in order. When there isn't enough room for the whole item in the next available cell, it tries the following cell, moving to the start of a new row if it doesn't fit in the final available cell of the current row (similar to floats). This method may leave empty cells, as shown in the figure. By comparison, the dense flow example on the right is all filled in, and if you look at the numbering, you can see that putting items in the first available space where they fit makes them end up out of order. Note that dense flow doesn't always result in a completely filled-in grid like in this figure, but it is likely to have fewer holes and be more compact than the default mode.

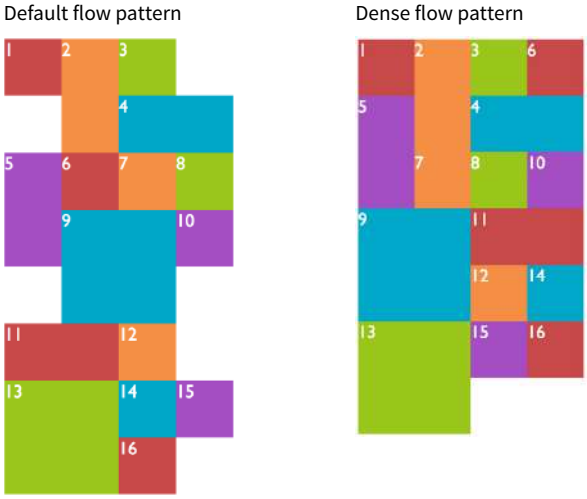


FIGURE 18-7. Comparison of default and dense auto-flow modes

You only know a handful of grid properties so far, but it's not too soon to play around with implicit grids in EXERCISE 18-1 and get a feel for default grid behavior. You'll also get a sneak peek at a few properties like `grid-template-columns` and `gap` that I'll cover in more detail later in the chapter.

EXERCISE 18-1. Working with an implicit grid

In this exercise, we'll take advantage of default grid behavior to align a list of terms and definitions into neat rows and columns. The starter document, *implicit-grid.html*, is available with the materials for this chapter at *learningwebdesign.com/6e/materials*. The content of the starter document is a list of cooking terms and their definitions marked up as a "definition list" (`dl`). There is also an embedded style sheet that has some basic styles for background colors, fonts, padding, and margins.

Start by opening the document in a browser. You should see alternating rows of terms and definitions, extending the width of the browser window. Note also that the elements have small number labels so you can see the order in which they are placed more easily.

1. Start by making the `dl` a grid container. Because the `dt` and `dd` elements are all direct children of `dl`, they all become grid items.

   ```
   dl {
     display: grid;
   }
   ```

   If you were to reload the file in the browser now, you wouldn't see much difference because all the elements are still stacked up in one column (sorry to ruin the surprise). Let's add a few more properties to make this more exciting.

2. Use the `grid-template-columns` property to give the grid two columns, one 8em wide and the other 1fr (fractional unit), which makes the second column fill the remaining width of the container (similar to `flex-grow`). Add some space between grid cells with the `gap` property. We'll talk more about these new properties and the `fr` unit later in the chapter, but for now, it should be clear what they do. After you've added the new declarations, save your document and take a look (FIGURE 18-8, left).

   ```
   dl {
     display: grid;
     grid-template-columns: 8em 1fr;
     gap: .5em;
   }
   ```

   The grid algorithm has created just enough rows to hold all of our `dt` and `dd` grid items, and automatically placed them in the cells across the rows. And we accomplished that with just two declarations! (`gap` is optional, but it is helpful to visualize the cells.) The numbers in each element reveal the order in which they were placed.

EXERCISE 18–1. Continued

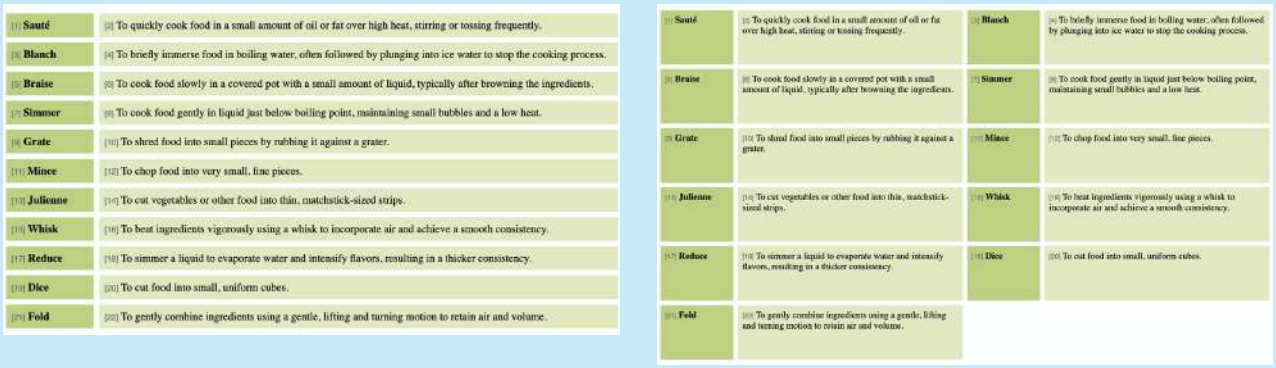

[1] Sauté
[2] To quickly cook food in a small amount of oil or fat over high heat, stirring or tossing frequently.
[3] Blanch
[4] To briefly immerse food in boiling water, often followed by plunging into ice water to stop the cooking process.
[5] Braise
[6] To cook food slowly in a covered pot with a small amount of liquid, typically after browning the ingredients.
[7] Simmer
[8] To cook food gently in liquid just below boiling point, maintaining small bubbles and a low heat.
[9] Grate
[10] To shred food into small pieces by rubbing it against a grater.
[11] Mince
[12] To chop food into very small, fine pieces.
[13] Julienne
[14] To cut vegetables or other food into thin, matchstick-sized strips.
[15] Whisk
[16] To beat ingredients vigorously using a whisk to incorporate air and achieve a smooth consistency.
[17] Reduce
[18] To simmer a liquid to evaporate water and intensify flavors, resulting in a thicker consistency.
[19] Dice
[20] To cut food into small, uniform cubes.
[21] Fold
[22] To gently combine ingredients using a gentle, lifting and turning motion to retain air and volume.

FIGURE 18-8. The resulting two-column and four-column grids for the definition list

[1] Sauté
[2] To quickly cook food in a small amount of oil or fat over high heat, stirring or tossing frequently.
[7] Simmer
[8] To cook food gently in liquid just below boiling point, maintaining small bubbles and a low heat.
[13] Julienne
[14] To cut vegetables or other food into thin, matchstick-sized strips.
[19] Dice
[20] To cut food into small, uniform cubes.
[3] Blanch
[4] To briefly immerse food in boiling water, often followed by plunging into ice water to stop the cooking process.
[9] Grate
[10] To shred food into small pieces by rubbing it against a grater.
[15] Whisk
[16] To beat ingredients vigorously using a whisk to incorporate air and achieve a smooth consistency.
[21] Fold
[22] To gently combine ingredients using a gentle, lifting and turning motion to retain air and volume.
[5] Braise
[6] To cook food slowly in a covered pot with a small amount of liquid, typically after browning the ingredients.
[11] Mince
[12] To chop food into very small, fine pieces.
[17] Reduce
[18] To simmer a liquid to evaporate water and intensify flavors, resulting in a thicker consistency.

FIGURE 18-9. When the direction is set to `column`, items flow in from top to bottom and new columns are added to accommodate them.

3. If we add a third value to the `grid-template-columns` property (`8em 1fr 1fr`), the grid will have three columns. You can give that a try and take a look. You will see that the items fill the grid in sequential order, but the three-column arrangement doesn't make sense for this content. We're going to add a fourth column, this time alternating 8em and 1fr widths and specifying a height of 6em for the automatically generated rows (`grid-auto-rows`). The four-column grid in FIGURE 18-8 (shows the result. Note that we didn't need to tell the grid how many rows to make; it figured it out for us. That's the useful feature of implicit grids.

```
dl {
  display: grid;
  grid-template-columns: 8em 1fr 8em 1fr;
  grid-auto-rows: 6em;
  gap: .5em;
}
```

4. The last thing we're going to do with this grid is change the flow direction to "column" with the `grid-auto-flow` property. Delete the `grid-template-columns` and `grid-auto-rows` declarations from the previous step to clear the way for this new orientation. (You could also wrap them in `/*` and `*/` to hide them in a comment.)

   Instead of setting column widths, this time we'll set a sequence of alternating row *heights*. The algorithm places the items in columns and automatically creates new columns until all of the items are accommodated. FIGURE 18-9 shows the result. Again, the numbers reveal the order in which they are placed in the implicit grid.

```
dl {
  display: grid;
  grid-auto-flow: column;
  grid-template-rows: 3em 1fr 3em 1fr 3em 1fr;
  gap: .5em;
}
```

EXPLICIT GRIDS

In the implicit grid examples, we set up a few column widths and let the browser do the rest. Now we'll look at how to specify a number of rows *and* columns, thus creating an explicit grid. The majority of this section will be reviewing the many options available for setting the size of tracks.

Explicit grids are often used for creating the alignment structure for an entire page, so in this section, we'll be working on a grid that has areas for a header, main content, two sidebars, and a footer (see **Note**). I've made a quick sketch of how I'd like the final grid to look (FIGURE 18-10). A sketch is a good first step for working with grids. From my sketch, I can see that the layout requires three row tracks and three column tracks, even though some of the content areas will eventually span over more than one cell.

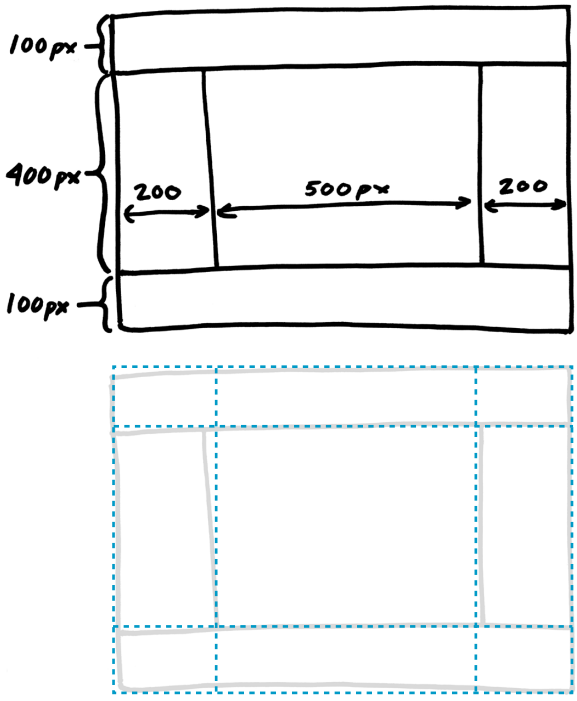


FIGURE 18-10. A rough sketch for our simple grid-based page layout. The dotted lines in the bottom image show how many rows and columns the grid requires to create the layout structure.

NOTE

You may be thinking that this page layout, with its header, footer, and three columns, looks like the one we made using Flexbox in FIGURE 17-26*. And you're right! It just goes to show that there may be several solutions for achieving an intended result.*

Defining Grid Tracks

To set up the grid, specify the height of each row and the width of each column with the template properties, `grid-template-rows` and `grid-template-columns`, which get applied to the container element.

grid-template-rows
grid-template-columns

Values: none | *list of track sizes and optional line names*

Default: none

Applies to: Grid containers

Inherits: No

The value of the **grid-template-rows** property is a list of the *heights* for each row track in the grid. The value of **grid-template-columns** is a list of the *widths* for each column track. As you've already seen in examples in this chapter, the number of track sizes provided determines the number of rows or columns. For example, if you provide four lengths for **grid-template-columns**, you get a grid that is initially divided into four columns.

Sticking with our five-item grid example, I've added template properties to divide the **#layout** container into three columns and three rows with the pixel dimensions I designated in the original sketch (FIGURE 18-11):

```
#layout {
  display: grid;
  grid-template-rows: 100px 400px 100px;
  grid-template-columns: 200px 500px 200px;
}
```

Let's see what happens if we do a quick check of the grid so far in the browser. As expected, the grid items autoflow sequentially into the available grid cells (FIGURE 18-11). The image on the right reveals the underlying grid structure, which would not be seen when the grid displays in the browser.

Because there are only five child elements in the **#layout div**, only the first five cells are filled, and the third row remains empty. It is worth pointing out that the structure established for explicit grids is independent of the number

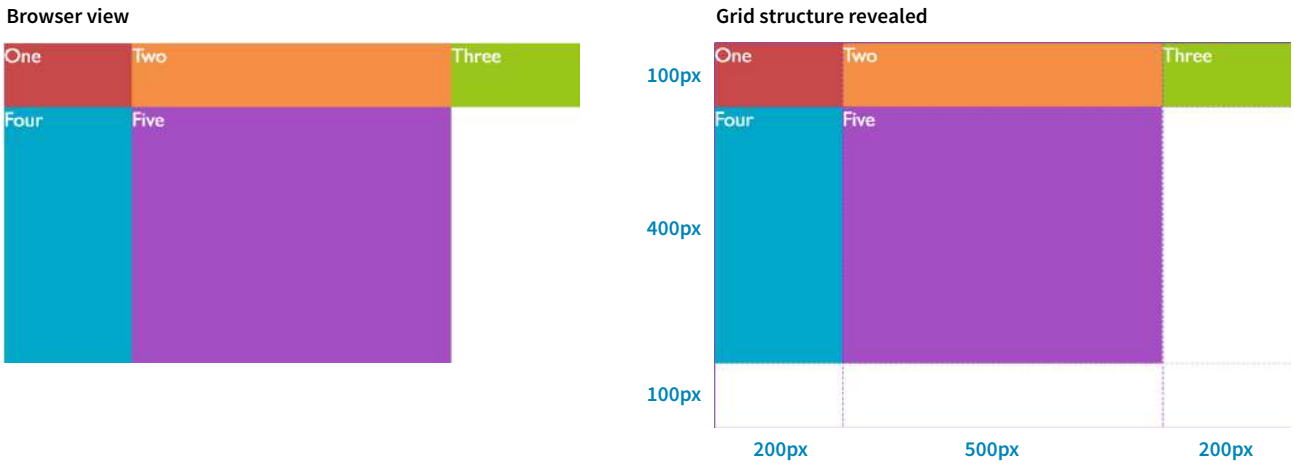


FIGURE 18-11. By default, grid items flow into the grid cells by rows.

of items in the container. In this example, the browser filled the first five of the nine cells, but (as you can see when the structure is revealed) preserved the rest. If you have a design that uses a 24-cell grid, you could deliberately place only a few items in the grid, leaving the remaining cells as "whitespace." That's the beauty of grid layout.

If the number of items *exceeds* the number of cells in the grid (for example, if there had been 14 child elements in our `#layout div` instead of 5), can you guess what happens? The grid algorithm will add as many new rows as necessary until every last item is placed! The added rows are sized according to the `grid-auto-rows` value, if provided; otherwise, they will be as tall as necessary to accommodate the content. You may run into scenarios in which an explicit grid with an implicit grid extending from it is a desired feature. For now, keep in mind that if you didn't make enough cells for your content, the browser will do it for you.

If you don't make enough cells for your content, the browser will do it for you.

Specifying Track Size Values

All of the track sizes in the previous example were specified in straightforward pixel lengths, but fixed sizes are just one of many track size options. The Grid Layout module provides a *whole bunch* of ways to specify track sizes, including old standbys like CSS length units (e.g., pixels, rems, or ems) and percentage values as well as some grid-specific values, including:

- The fractional unit (`fr`)
- The `repeat()` function
- The `minmax()` function
- Content-based values: `auto`, `min-content`, `max-content` (see **Note**)

NOTE

The `fit-content()` *function also bases track size on the content, but I will not be covering it in this chapter. See the spec (w3.org/TR/css-grid-2/) for details.*

Fractional units (fr)

You used an `fr` value in EXERCISE 18-1, but it bears a little more explanation. The grid-specific fractional unit (`fr`) allows authors to create track widths that expand and contract depending on available space, similar to the `flex-grow` property for Flexbox.

Going back to our explicit grid example, if we change the width of the middle column from 500px to 1fr, the browser assigns all leftover space (after the 200-pixel column tracks are accommodated) to that column track. The middle column now flexes to adapt to the available width of the viewport (FIGURE 18-12):

```
#layout {
  display: grid;
  grid-template-rows: 100px 400px 100px;
  grid-template-columns: 200px 1fr 200px;
}
```

■ AT A GLANCE

Track Size Values

The Grid Layout specification allows the following values for the `grid-template-*` properties:

- CSS length units (such as `px`, `em`, or `rem`)
- Percentage values (`%`)
- Fractional units (`fr`)
- `repeat()`
- `minmax()`
- `auto`
- `min-content`, `max-content`
- `fit-content()`

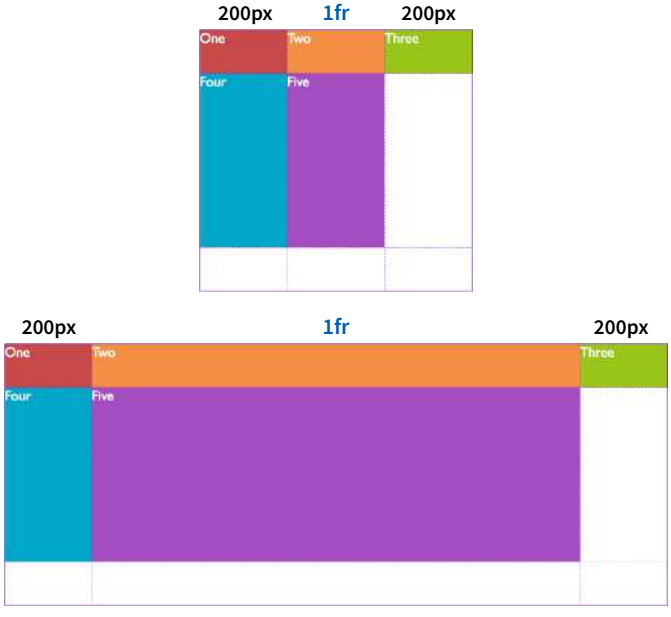


FIGURE 18-12. When the middle column has a track size of 1fr, it takes up the remaining space in the browser window and flexes to adapt to the browser width.

The `fr` unit is great for combining fixed and flexible track widths, but we could also use all `fr` units to give all the columns proportional widths. When there are multiple tracks with `fr` units, the browser adds up the units and divides the available space into that many portions. The portions are assigned based on the number of units specified. For example, all of the column widths in the following grid sample flex according to the available browser width, but the middle column will always be twice the width of the side columns:

```
grid-template-columns: 1fr 2fr 1fr;
```

Minimum and maximum size range

Another useful value for track sizes is the `minmax()` function, which provides flexibility but allows you to set limits. You can constrict the size range of a track by setting its minimum and maximum widths using the `minmax()` function in place of a specific track size. The minimum and maximum track sizes (in that order) are placed within parentheses, separated by a comma:

```
grid-template-columns: 200px minmax(10em, 30em) 200px;
```

This rule sets the middle column to a width that is at least 10em but never wider than 30em.

WARNING

`fr` units are not permitted as the minimum value in a `minmax()` statement.

Repeating track sizes

Say you have a grid that has 10 columns with alternating column widths, like so:

```
grid-template-columns: 20px 1fr 20px 1fr 20px 1fr 20px 1fr 20px 1fr;
```

That's kind of a bummer to have to type out (I know, I just did it), so the folks at the W3C have provided a nice shortcut in the form of the **repeat()** function. In the previous example, the pattern "20px 1fr" repeats five times, which can be written as follows:

```
grid-template-columns: repeat(5, 20px 1fr);
```

Much better, isn't it? The first number indicates the number of repetitions, and the track sizes after the comma provide the pattern. You can use the **repeat()** function in a longer sequence of track sizes—for example, if those 10 columns are sandwiched between two 200-pixel-wide columns at the start and end:

```
grid-template-columns: 200px repeat(5, 20px 1fr) 200px;
```

We could have used the **repeat()** function in step 4 of EXERCISE 18-1 to simplify the row track height listing (look back to compare):

```
dl {
  display: grid;
  grid-auto-flow: column;
  grid-auto-rows: repeat(3, 3em 1fr);
  gap: .5em;
}
```

WARNING

You can only use one `repeat()` *notation for a given declaration. You can use* `minmax()` *inside* `repeat()`*, but* `fr` *units and content-based keywords may only be used as the max value.*

Content-based sizing

The **min-content**, **max-content**, and **auto** values size the track based on the size of the content within it. FIGURE 18-13 shows the result of using **min-content** and **max-content** for a cell containing text.

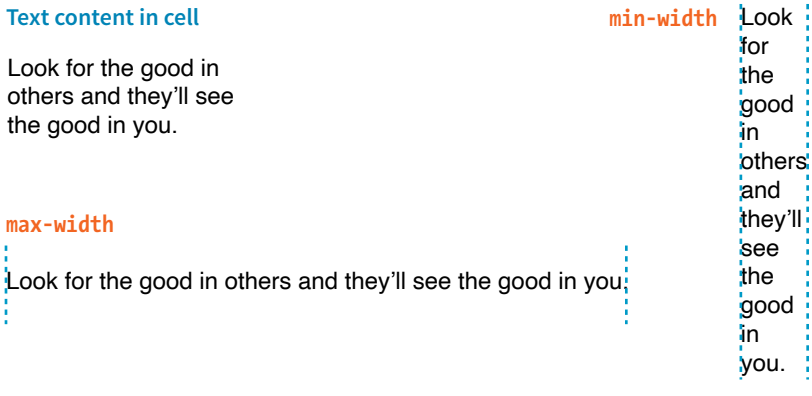

Text content in cell
Look for the good in others and they'll see the good in you.
max-width
Look for the good in others and they'll see the good in you.
min-width
Look for the good in others and they'll see the good in you.

FIGURE 18-13. The `min-content` and `max-content` track sizing values

min/max-content

The `min-content` value is the *smallest* the track can get without overflowing (by default, unless overridden by an explicit `min-width`). It is equivalent to the "largest unbreakable bit of content"—in other words, the width of the longest word or widest image. It's unlikely to be useful for items that contain normal paragraphs, but it may be useful in some cases where you don't want the track larger than it needs to be. This example establishes three columns, with the righthand column sized just wide enough to hold the longest word or image:

```
grid-template-columns: 50px 1fr min-content;
```

The `max-content` property allots the maximum amount of space needed for the content, even if that means extending the track beyond the boundaries of the grid container. When it's used as a column width, the column track will be wide enough to accommodate the widest content in that track *without* line wrapping. That means if the content is a paragraph, the track will be wide enough to contain all of the text set on one line. This makes `max-content` more appropriate for short phrases or navigation items whose text you don't want to wrap (although `auto` may work better, because it allows wrapping if there's not enough room).

The auto keyword

Using the `auto` keyword for a track size is basically like handing the keys over to the browser. In general, it causes the track to be sized large enough to accommodate its content, while taking into consideration what other restrictions are in place. If you want to size a track based on its content, but you're not sure which keyword to use, start with `auto`.

If you want to size a track based on its content, but you're not sure which keyword to use, start with auto.

As a keyword on its own, it functions similarly to `minmax(min-content, max-content)`, allowing the track to be as narrow as it can be without anything overflowing but to grow to fit its content without wrapping if there's enough space.

When you use the `auto` keyword in the `minmax()` function, it behaves very similarly to either `min-content` or `max-content`, depending on whether you put it in the minimum or maximum slot. An `auto` maximum value allows extra space to be added to the track based on alignment settings (not the case for `max-content`). As a minimum, it uses the specified `min-width` or `min-height` of an item (if any) instead of its `min-content` size.

That's a wrap on setting up explicit grids and track size values. I'm thinking that it's a good time for you to put all these grid setup styles to use in EXERCISE 18-2.

A Brief Introduction to subgrid

Earlier in the chapter, I pointed out that only direct children of a grid container element become grid items. It's possible to make a child element display as a grid as well, thus nesting a grid within a grid, but doing that alone does not make the tracks for the child item line up with its parent's. To achieve that, CSS Grid Layout Module Level 2 introduced the `subgrid` value, which applies the parent grid's track measurements to the nested grid item.

To use it, first make the child element display as a grid (`display: grid`). If you want the nested grid to use the same column tracks as its parent, set the `grid-template-columns` property to `subgrid`, and it will inherit those widths. Similarly, if you want the nested grid to follow the alignment of its parent's rows, use `subgrid` as the value for `grid-template-rows`. With `subgrid`, the nested grid inherits any space between tracks set with the `gap` property as well.

The full story on using `subgrid` requires more discussion than I have room for here, but you can get up to speed with the `subgrid` reference page on MDN Web Docs (*oreil.ly/7n8Xr*). You will also find a number of good video tutorials on YouTube.

Subgrids are one of the more recent additions to the Grid Layout toolkit, so browser support is a little less widespread than for other properties. Be sure to check *CanIUse.com* for updated browser support statistics.

EXERCISE 18-2. Setting up a grid

In this exercise, we'll set up the grid template for the page shown in FIGURE 18-14. We'll place the grid items into the grid in EXERCISE 18-3, so for now just pay attention to setting up the rows and columns.

This page is similar to the bakery page we've been working on, but it has some additional elements and whitespace to make things interesting. The starter document, *grid.html*, is provided with the materials for this chapter. Open it in a text or code editor, and you'll see that all of the styles affecting the appearance of each element are already provided.

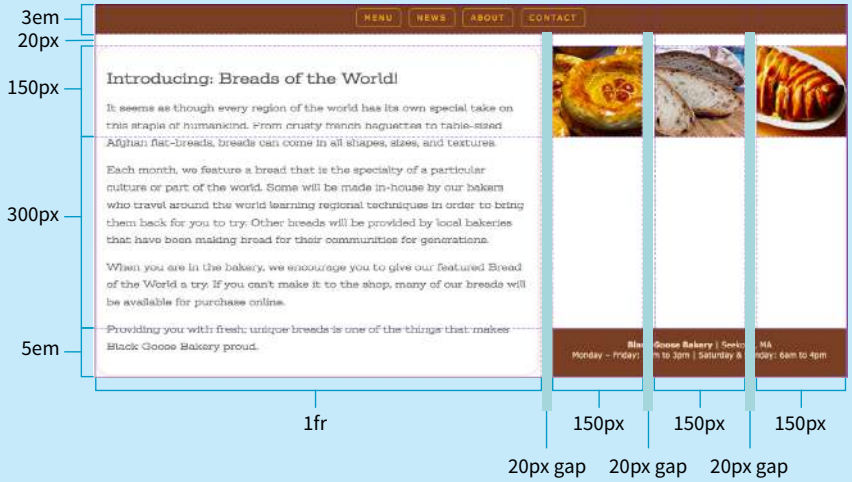

3em
20px
150px
300px
5em
MENU
NEWS
ABOUT
CONTACT
Introducing: Breads of the World!
It seems as though every region of the world has its own special take on this staple of humankind. From crusty french baguettes to table-sized Afghan flat-breads, breads can come in all shapes, sizes, and textures.
Each month, we feature a bread that is the specialty of a particular culture or part of the world. Some will be made in-house by our bakers who travel around the world learning regional techniques in order to bring them back for you to try. Other breads will be provided by local bakeries that have been making bread for their communities for generations.
When you are in the bakery, we encourage you to give our featured Bread of the World a try. If you can't make it to the shop, many of our breads will be available for purchase online.
Providing you with fresh, unique breads is one of the things that makes Black Goose Bakery proud.
1fr
150px
150px
150px
20px gap
20px gap
20px gap

FIGURE 18-14. The "Breads of the World" page that we'll create using Grid Layout

1. Start by turning the containing element, **`#layout div`**, into a grid container by setting its display mode to **`grid`**:

```
#layout {
  ...
  display: grid;
}
```

2. FIGURE 18-14 shows the row and column tracks required to accommodate the content in the desired layout. Start by defining the rows as specified in the sketch, using the **`grid-template-rows`** property. There should be five values, representing each of the five rows. (Spoiler alert: we'll be tweaking these values when we get to the next exercise. This is just a starting point.)

```
#layout {
  ...
  display: grid;
  grid-template-rows: 3em 20px 150px 300px 5em;
}
```

3. Do the same for the four columns. Because we want the width of the main text column to grow and shrink with the available space, we'll specify its width in fractional units (1fr). For the remaining three columns, create 150-pixel-wide cells to hold three images. The 20-pixel spaces between the columns could be included in the column track count, but because they are a consistent width, and because we will not be placing any content in those spaces, they are better handled with the **`gap`** property. We'll take care of gaps in the next step.

EXERCISE 18–2. Continued

You can write out all the column tracks like this:

```
grid-template-columns: 1fr 150px 150px 150px;
```

However, because it is a repeating pattern, you could also use the `repeat()` function:

```
grid-template-columns:  1fr repeat(3, 150px);
```

4. Now we'll add the gaps between columns. I'll get to the formal introduction of the gap-related properties in a moment, but they are pretty intuitive, and you've already used one in EXERCISE 18-1. For this grid, we'll add a 20-pixel gap between columns by adding this declaration to the style rule for `#layout`:

   ```
   column-gap: 20px;
   ```

5. Finally, let's assign names to the grid lines that border the grid area where the `main` content element should appear. (I'll formally introduce line names in the following section, but let's get this done while we have the document open. Copy the code example and stay tuned for more explanation.) The names give us an intuitive option for placing items later. The main area starts at the third row track, so assign the name "main-start" to the grid line just before the third row track measurement:

   ```
   grid-template-rows: 3em 20px [main-start] 150px 300px 5em;
   ```

   The main area extends down to the bottom of the grid, so assign the name "main-end" after the last row track:

   ```
   grid-template-rows: 3em 20px [main-start] 150px 300px 5em [main-end];
   ```

6. Now do the same for the grid lines that mark the boundaries of the column track where the main content goes:

   ```
   grid-template-columns: [main-start] 1fr [main-end] repeat(3, 150px);
   ```

Save your work and look at it in the browser (FIGURE 18-15). Because we haven't yet specified where the grid items go, the grid algorithm autoplaced content elements into the cells sequentially, making the mess you see in the figure. However, the grid overlay reveals that the structure of the grid looks solid. Save the file and hold on to it until the next exercise.

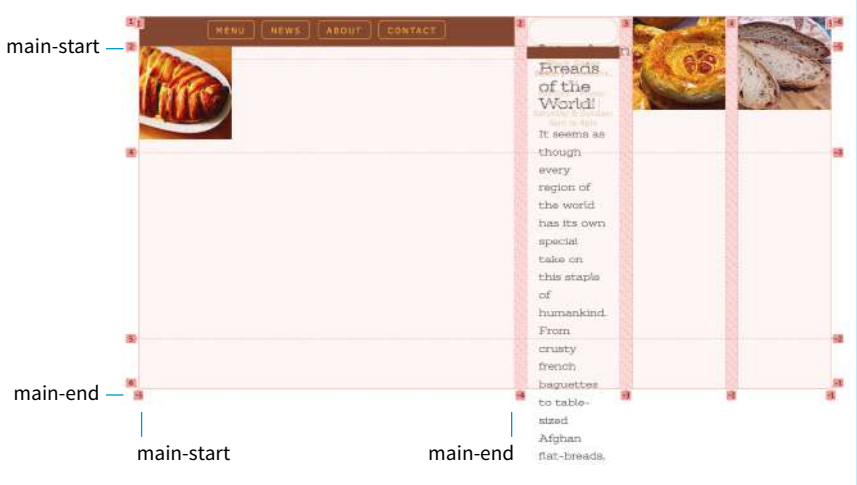


FIGURE 18-15. The grid items are not placed correctly yet, but the Grid Inspector (introduced in the next section) shows that the grid is set up correctly.

PLACING ITEMS ON THE GRID

In all of the examples so far, we've relied on the grid algorithm's autoplacing behavior to put the grid items into cells sequentially. That method certainly has its place, but the real power of Grid Layout lies in the ability to place an item into the specific cell or grid area of your choice. This is where Grid Layout may start to feel a little complicated, because there are so many options for doing the same thing. I'll introduce you to the various item-placing methods and you can choose the ones that are the most comfortable for you.

There are two general approaches to placing items in a grid:

- By specifying the grid *lines* at which the item should start and end, either by line number or name
- By targeting a named *area* of the grid

Positioning Items Using Grid Lines

I introduced the term "grid line" briefly when I introduced the parts of a grid, but now we're going to put them to use as reference points for placing grid items right where we want them to go. You can refer to a specific grid line by its numerical position in the grid or by a name you assign it when setting up the grid tracks.

Using automatic line numbering

Behind the scenes, the grid algorithm assigns a number to every line in the grid that you can use to specify the start and end point for an item. Before we get to code examples, this is a good opportunity to introduce a great tool for visualizing grid line numbering and grid structure. Chrome, Safari, Edge, and Firefox each include a Grid Inspector as part of their Developer Tools (we looked at those tools briefly in **Chapter 11, Introducing Cascading Style Sheets**).

FIGURE 18-16 shows how the example grid looks in Chrome's Grid Inspector, with the content hidden for clarity. To access the inspector in Chrome, select View → Developer → Developer Tools. Choose the Element Inspector so you get a view of the document's HTML structure and styles. When you select the element that has been turned into a grid container (`#layout div` in our example), the word "grid" appears next to the opening tag. Clicking the grid button displays the Grid Inspector overlay on the page preview.

The vertical grid line on the left edge of the grid is assigned the number 1, and lines are numbered sequentially from there, ending with the grid line on the right edge of the grid (see **Note**). The numbering of horizontal grid lines (row tracks) begins at the top of the grid and ends with the bottom grid line.

NOTE

The numbering direction reflects the writing direction of the document. Because my document is in English, numbering goes from left to right and top to bottom, but the direction would change for languages with alternative writing directions.

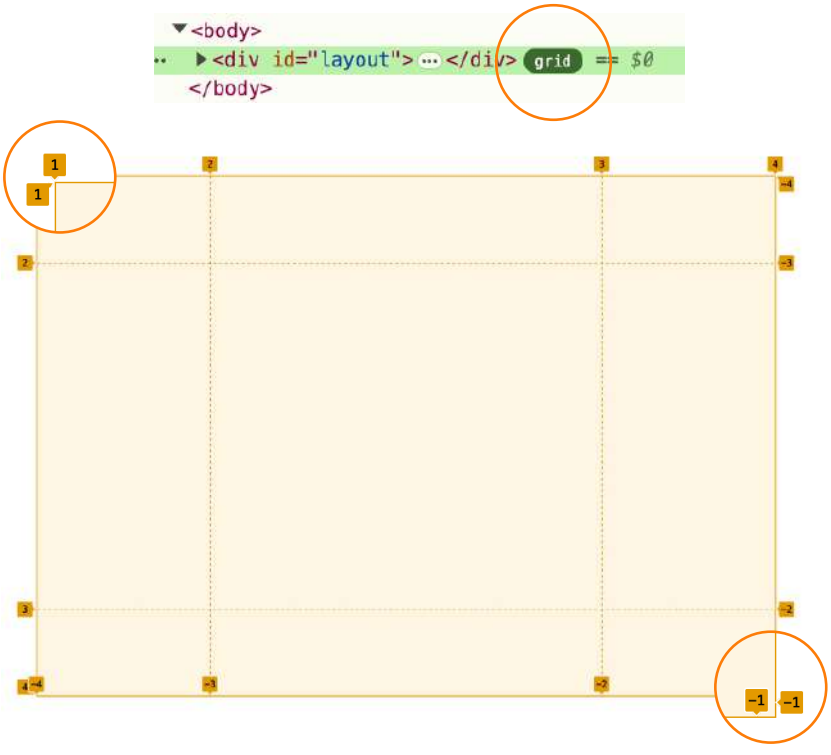


FIGURE 18-16. Chrome's Grid Inspector reveals the numbers assigned to each line in the sample grid.

The lines are numbered from the end of tracks as well, starting with –1 and counting back from there (–2, –3, etc.), as shown along the bottom and right edges in the Grid Inspector view in FIGURE 18-16. In the example grid structure, the vertical grid line at the right edge of the grid could be targeted in a property either by the number 4 or by –1 (note the numbers in the overlay). Being able to target the end of a row or column without counting lines (or even knowing how many rows or columns there are) is a handy feature. You'll come to love that –1.

> **TIP**
>
> The Chrome Grid Inspector can also show the track sizes and area names in the overlay. To turn on those options, open the Layout panel.

Assigning line names

If you don't like to keep track of numbers, you can also assign lines names that may be more intuitive. You assign names to lines when you set up the track sizes with one of the `grid-template-` properties. Write each line name within square brackets (`[]`) in the position it appears relative to its neighboring tracks (you may have already tried this in EXERCISE 18-2).

In the following example, I've assigned names that correspond to how I will be using each track in the final page:

```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [content-start] 400px
[footer-start] 100px [footer-end];
  grid-template-columns: [ads] 200px [main] 500px [links] 200px;
}
```

Based on this example, the grid line at the top of the grid can now be referred to as "header-start," 1, or –4. I could also name the line that comes after the first row track "header-end", even though I've already given it the name "content-start". To give a line more than one name, just include all the names in the brackets, separated by spaces:

```
grid-template-rows: [header-start] 100px [header-end content-start]
400px [footer-start] 100px [footer-end];
```

It is common for each grid line to end up with multiple names and numbers, and you can choose whichever is the easiest to use.

Properties for positioning

Positioning properties apply to the grid item elements being placed.

Now that you know how to reference grid lines, let's look at the properties that use them. One method for describing a grid item's location on the grid is to specify the four lines bordering the target grid area using the **-start** and **-end** properties. These are the first properties we've seen so far that get applied to the individual grid items rather than the grid container. That makes sense, because we are manipulating the position of one item at a time.

`grid-row-start`
`grid-row-end`
`grid-column-start`
`grid-column-end`

Values: auto | *grid line* | span *number* | span *'line name'* | *number 'line name'*

Default: auto

Applies to: Grid items

Inherits: No

This set of properties provides a straightforward way to describe an element's position on the grid by identifying either the name or the number of the grid line on each border. As an alternative, you can provide just one line identifier and tell the item to "span" a certain number of cells (we'll look at spanning in a moment). By default, an item occupies one track width, which is what you get with the **auto** keyword.

Getting back to our five-item example, I would like the first item to go in the top row and span across all three columns (FIGURE 18-17)—or in other words, to fill the area corresponding to the header in our sketch (FIGURE 18-10).

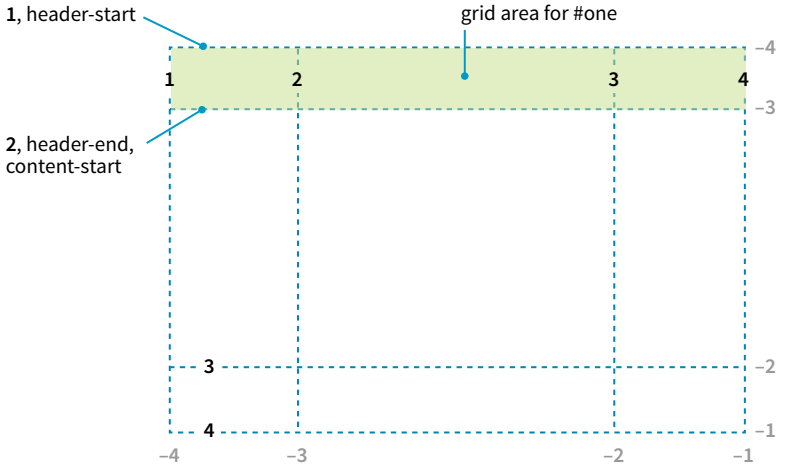


FIGURE 18-17. Positioning a grid item across the top row track in our sample grid

One way to do this is to use the four line start/end properties and identify lines by their numbers, like so:

```
#one {
  grid-row-start: 1;
  grid-row-end: 2;
  grid-column-start: 1;
  grid-column-end: 4;
}
```

For **grid-row-start**, the 1 value refers to the first (top) line of the grid container. For **grid-row-end**, the 2 value refers to the line below the first row track.

For **grid-column-start**, 1 refers to the first line on the left edge of the container, and the value 4 for **grid-column-end** identifies the fourth and last line on the right edge of the container.

Here's one more for good measure. This style declaration positions the **#four** item element in the right column, as shown in FIGURE 18-10:

```
#four {
  grid-row-start: 2;
  grid-row-end: 3;
  grid-column-start: 3;
  grid-column-end: 4;
}
```

NOTE

If you omit a start or end line, the area will be one track wide (the default, **auto***).*

Using negative line numbers

Remember how grid lines are also numbered in the opposite direction, starting at –1? We can use that here. We could specify the **grid-column-end** for **#one** as –1, and it would be the same as 4. In fact, this method has the advantage of guaranteeing to span to the end of the grid and avoids the risk of miscounting:

```
#one {
  grid-row-start: 1;
  grid-row-end: 2;
  grid-column-start: 1;
  grid-column-end: -1;
}
```

We could also use the named lines we set up here. These row values are interchangeable with the previous example:

```
#one {
  grid-row-start: header-start;
  grid-row-end: header-end;
  ...
}
```

If you omit the end line declaration, the row will be one track high (the default). In this case the header area is only one row high, so omitting the end declaration altogether would be one more way to achieve the effect we want.

> **TIP**
>
> If you need to span to the last grid line in a row or column, use the value –1 and save yourself some counting. Even if the number of rows or columns changes down the line, –1 will always select the last line, so you won't need to renumber.

The span value

Ready for yet another option? We can tell the item what line to start on, but instead of providing an end line, we can use the **span** keyword to specify how many tracks to span over. In this example, the item **#one** starts at the left edge of the track (line 1) and spans over three columns, effectively ending at line 4:

```
#one {
  ...
  grid-column-start: 1;
  grid-column-end: span 3;
}
```

Spans can work in reverse as well. If you provide only an end line, the span searches backward toward the start of the track. The following styles have the same effect as our previous examples because they define the target area by its end line at the far right of the grid and span back three columns to the beginning:

```
#one {
  ...
  grid-column-start: span 3;
  grid-column-end: -1;
}
```

The Grid Layout module includes a few more uses of **span** that you can explore on your own.

Shorthand placement properties

If four declarations feels like too many, use the shorthand **grid-row** and **grid-column** properties instead.

grid-row
grid-column

Values: *start line* / *end line*

Default: See individual properties

Applies to: Grid items

Inherits: No

These properties combine the **-start** and **-end** properties into a single declaration. The start and end line values are separated by a slash (**/**). With the shorthand, you can shorten any of the previous examples to two declarations. All of these rules do the same thing in different ways:

```
#one {
  grid-row: 1 / 2;
  grid-column: 1 / 4;
}

#one {
  grid-row: header-start / header-end;
  grid-column: 1 / -1;
}

#one {
  grid-row: 1 / 2;
  grid-column: span 3 / -1;
}
```

> **TIP**
>
> To create a grid area that spans the entire width of a grid, use this nice shortcut:
>
> ```
> grid-column: 1 / -1
> ```
>
> Similarly, applying the 1 / –1 value to **grid-rows** creates an area that spans the entire height of the grid.

Positioning Items Using Area Names

Positioning items with grid lines is a robust and flexible method, but the Grid Layout module provides another approach that may be even more intuitive for some purposes. You just have to be willing to do some planning and interesting configuring up front when setting up the template for the grid.

Assigning named grid areas

In addition to naming grid lines, you can assign names to *areas* of the grid. Remember that a grid area is made up of one or more cells in a rectangle (no L-shapes or other nonrectangular collections of cells). Naming grid areas is a little funky to implement, but provides nice shortcuts when you need them.

To assign names to grid areas, use the **grid-template-areas** property in addition to **grid-template-rows** and **grid-template-columns**. All of these properties apply to the grid container.

grid-template-areas

Values: none | *series of area names*

Default: none

Applies to: Grid containers

Inherits: No

The value of this property is a list of names provided for every cell in the grid, listed row by row, with each row in quotation marks. When neighboring cells share a name, they form a grid area with that name. This will be made clearer with an example.

As FIGURE 18-18 shows, I've given names to areas in the example grid we've been working on so far.

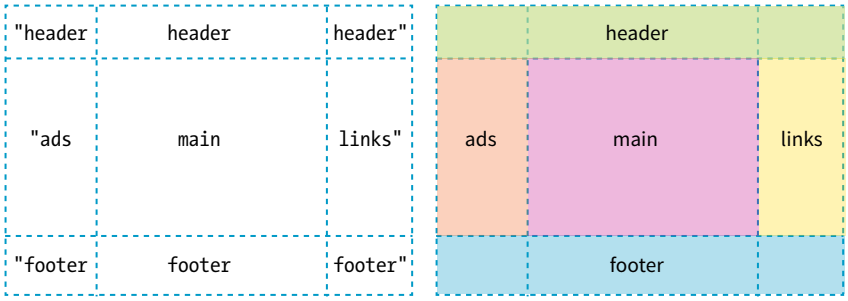


FIGURE 18-18. When neighboring cells have the same name, they form a named area that can be referenced later.

In the **grid-template-areas** value, there is a cell name for each of the nine cells as they appear in each row (compare the value to the sketch in the figure). The row cell lists don't need to be stacked as I've done here, but many developers find it helpful to line up the cells by name using character spaces to better visualize the grid structure:

```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [content-start] 400px
[footer-start] 100px;
  grid-template-columns: [ads] 200px [main] 1fr [links] 200px;
  grid-template-areas:
    "header  header  header"
    "ads     main    links"
    "footer  footer  footer";
}
```

Be aware that the track sizes are still coming from the **grid-template-columns** and **grid-template-rows** properties. The **grid-template-areas** property simply assigns names to the areas, making it easier to plop items in them later. If there are three columns in the grid, there must be three names provided for

NOTE

Be sure that you place the cell names in a way that forms rectangles when they combine to identify a named area. No L-shapes or fragments.

each row. If you want to leave a cell unnamed, type one or more periods (`.`) in its place as a space holder so that every cell is still accounted for. Again, a sketch of your grid with the areas identified will make it easier to plan out the `grid-template-areas` value.

In our example, when the browser sees the word "header" three times next to one another, it creates an area equivalent to `grid-row: 1 / 2; grid-column: 1 / 4`. The "ads", "main", and "links" areas are made up of one grid cell each. The "footer" area works similarly to "header", but at the bottom of the grid. Assigning names to areas also automatically generates line names for the neighboring grid lines, as explained in the **"Bonus Grid Line Names"** sidebar.

Bonus Grid Line Names

When you give an area a name with `grid-template-areas`, as an added bonus, you get a set of automatically generated grid line names to go with it. For example, when you name an area "main", the left and top grid lines of that area are automatically named "main-start", and the right and bottom grid lines are named "main-end". You can use those line names when positioning items.

The inverse is true as well. If you explicitly assign line names "portal-start" and "portal-end" around an area, you can use the area name "portal" to assign content to that area later, even if you haven't defined it with `grid-template-areas`. You can keep this shortcut in mind when naming grid lines, but using it is not required.

This is a prime example of the flexibility and complexity of the Grid Layout module.

Positioning items by grid area name

Once you have set up the names of the grid areas, you can assign grid items to positions by area name using the `grid-area` property.

grid-area

Values:	*area name* | *1 to 4 line identifiers*
Default:	See individual properties
Applies to:	Grid items
Inherits:	No

The `grid-area` property points to one of the areas named with `grid-template-areas`. It can also point to an area name that is implicitly created when you name lines delimiting an area with the suffixes "-start" and "-end," as mentioned in the **"Bonus Grid Line Names"** sidebar. With this method, we can drop all of the grid items into the areas we set up with our template earlier (FIGURE 18-19):

```
#one   { grid-area: header; }
#two   { grid-area: ads; }
#three { grid-area: main; }
#four  { grid-area: links; }
#five  { grid-area: footer; }
```

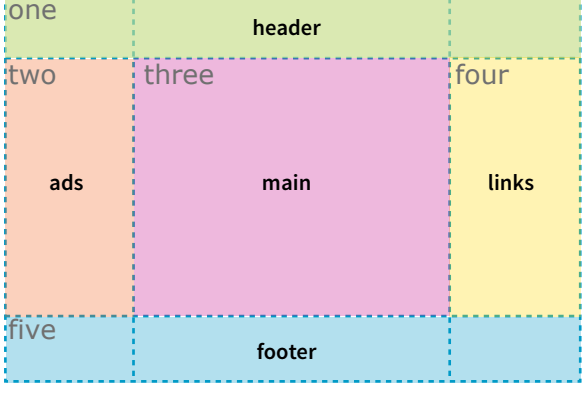


FIGURE 18-19. Assigning grid items by area names

How easy was *that*?! One benefit of using areas is that you can change the grid, and as long as you provide consistently named grid areas, the items will end up in the right place. There's no need to renumber lines in the style sheet.

Using grid-area with line numbers

You can also use **grid-area** to provide a list of four grid lines that define an area, separated by slashes. The order in which they appear is "row-start", "column-start", "row-end", and "column-end" (counterclockwise from the top). There are a lot of rules for what happens when you omit values, but we're not going to get into all those finer points here (you can refer to the spec). The **grid-area** declaration for the first grid item could be written like this to achieve the same result as previous examples:

```
#one {
  grid-area: 1 / 1 / 2 / 4;
  /* row-start / column-start / row-end / column-end */
}
```

Now that you know the options for putting items where you want them to go, let's finish up the grid we've been working on in EXERCISE 18-3.

EXERCISE 18-3. Placing items on a grid

We have the grid set up for the "Breads of the World" page, so now it's time to place items into the correct grid areas by using line numbers and names.

I'm going to go through the steps quickly, but feel free to save the file and look at the page in a browser at any step along the way. Refer to the finished layout in FIGURE 18-20 for the final item positions and line number hints.

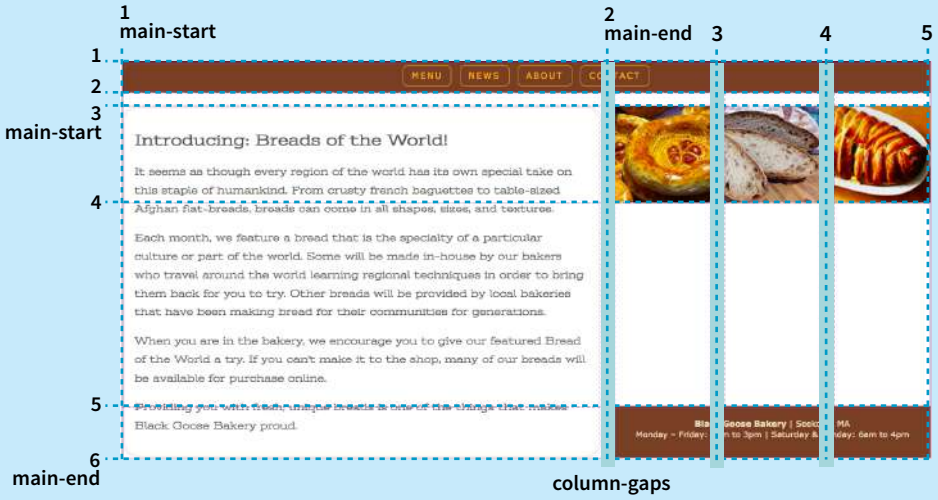

1
main-start
2
main-end
3
4
5
1
2
3
main-start
4
5
6
main-end
column-gaps
MENU
NEWS
ABOUT
Introducing: Breads of the World!
It seems as though every region of the world has its own special take on this staple of humankind. From crusty french baguettes to table-sized Afghan flat-breads, breads can come in all shapes, sizes, and textures.
Each month, we feature a bread that is the specialty of a particular culture or part of the world. Some will be made in-house by our bakers who travel around the world learning regional techniques in order to bring them back for you to try. Other breads will be provided by local bakeries that have been making bread for their communities for generations.
When you are in the bakery, we encourage you to give our featured Bread of the World a try. If you can't make it to the shop, many of our breads will be available for purchase online.
Black Goose Bakery proud.

FIGURE 18-20. The final "Breads of the World" grid layout

EXERCISE 18–3. Continued

1. Open *grid.html* in your text or code editor if it isn't open already. We'll start by placing the **nav** element into the first row of the grid, using the four grid line properties:

```
nav {
  grid-row-start: 1;
  grid-row-end: 2;
  grid-column-start: 1;
  grid-column-end: 5; /* you could also use -1 */
}
```

2. Now place the images in their positions on the grid. Start by putting the third figure (**#figC**) in its place in the far-right column by using the shorthand **grid-row** and **grid-column** properties. **#figC** goes between row grid lines 3 and 4 and column lines 4 and 5. For the columns, let's get fancy and try spanning one column in the reverse direction. Specifying the negative line number (–1) as the end point and providing a span value (1) targets the cell that starts at the last column line (the right edge of the grid) and spans back one track to line 4. It's just one of several options for placing this item.

```
#figC {
  grid-row: 3 / 4;
  grid-column: span 1 / -1;
}
```

   Next, position the **#figA** and **#figB** elements by using the **grid-area** property with four line values. Remember that the values go in the order top, left, bottom, right (counterclockwise around the area). This is an even more compact method, but of course, you could use the **grid-row-** and **grid-column-** properties instead if that feels more straightforward. It depends on personal preference and what makes sense for the grid structure you're working on.

```
#figA {
  grid-area: 3 / 2 / 4 / 3;
}
#figB {
  grid-area: 3 / 3 / 4 / 4;
}
```

3. We gave the grid lines around the main content area names when we set up this grid, so let's use them to place the **main** grid item:

```
main {
...
  grid-row: main-start / main-end;
  grid-column: main-start / main-end;
}
```

   Do you remember that when you name lines around an area ***-start** and ***-end**, it creates an implicitly named area *? Because we named the lines according to this syntax, we could also place the **main** element with **grid-area** like this:

```
main {
  ...
  grid-area: main;
}
```

4. Finally, we'll put the footer into its place. It starts at row grid line 5 and spans down one track. For columns, it starts at the second line and goes to the last. Here is one way to write those instructions. Can you come up with others that achieve the same result?

```
footer {
  ...
  grid-row: 5 / span 1;
  grid-column: 2 / -1;
}
```

5. Save your file and look at it in the browser. You may spot a problem, depending on the width of your browser window. When the browser is wide, the layout works fine, but when it is made narrower, the text in the **main** element overflows its cell. That's because the 300-pixel height we gave that row is not sufficient to hold the text when it breaks onto additional lines or is resized larger. We can fix that by changing the measurement of the fourth row track to **auto**. Then, the height of that row will always be at least big enough to hold the content. The **min-content** value would work as well, but **auto** is always the first value to try.

```
#layout {
  display: grid;
  grid-template-rows: 3em 20px [main-start] 150px →
auto 5em [main-end];
  ...
}
```

If you reload the page in the browser and resize the window a few times, you'll see that the text is always contained in its grid area, regardless of the width of the window. Everything should fall into place nicely, as shown in FIGURE 18-20.

You now have your first explicit grid layout under your belt. This exercise only gave you a taste of what Grid Layout can do, but we've covered the fundamentals of setting up a grid and placing items in it. You're off to a great start!

THE GRID SHORTCUT PROPERTY

For those who like to keep their style sheets as efficient as possible, the Grid Layout module provides the **grid** shortcut property. The syntax differs based on whether it is used for explicit or implicit grids.

grid Shorthand for Explicit Grids

Now that you've been introduced to all the ways to set up an explicit grid, including track size values and assigning line and area names, it's time to put them all together in the **grid** shorthand property. You can use this property to set values for **grid-template-rows**, **grid-template-columns**, and **grid-template-areas** with one style rule. Bear in mind that any properties you do not use will be reset to their defaults, as is the case for all shorthand properties.

grid

Values: none | *row info* / *column info*

Default: none

Applies to: Grid containers

Inherits: No

In **grid**, the row values and column values are separated by a slash, with the row values appearing first. This example is a very simplified view of the value structure:

```
grid: rows / columns
```

Here is the shorthand declaration for our example explicit grid with only row and column track information. I've omitted line and area names for now because it is easier to grasp the **grid** syntax without them (see **Note**); I'll add the names back in one step at a time:

```
#layout {
  display: grid;
  grid: 100px 400px 100px / 200px 1fr 200px;
}
```

Because the track values for **grid** tend to get quite long, it's common to move the track information to their own lines for clarity. You'll see why this is helpful soon. Here is the same **grid** declaration with stacked row and column values:

```
#layout {
  display: grid;
  grid:
    100px 400px 100px
    / 200px 1fr 200px;
}
```

NOTE

Not all grids use named lines and areas, so they are optional, of course.

To include custom line names, add the names in brackets (**[]**) next to their respective tracks, as we did in the named line examples earlier in this chapter. Let's add the column line names we established back into the column track listing:

```
#layout {
  display: grid;
  grid:
    100px 400px 100px
    /[ads] 200px [main] 1fr [links] 200px;
}
```

Including area names looks a little convoluted at first, but if you remember that you assign cell names row by row, it makes sense that they appear with the other row information, before the slash. The complete syntax for row values goes in this order:

```
[start line name] "area names" <track size> [end line name]
```

Repeat this structure for each row in the grid, simply listing them one after another with no special character separating rows. You can see how stacking values as we've done in the following example helps keep each row distinct. When the rows are done, add a slash, and list the column track information after it. Here's the complete declaration for our example grid using the **grid** shorthand:

```
#layout {
  display: grid;
  grid:
    [header-start]  "header  header  header" 100px [header-end]
    [content-start] "ads     main    links"  400px
    [footer-start]  "footer  footer  footer" 100px [footer-end]
    /[ads] 200px [main] 1fr [links] 200px;
}
```

This expands to the following:

```
#layout {
  display: grid;
  grid-template-rows: [header-start] 100px [header-end content-start]
400px [footer-start] 100px[footer-end];
  grid-template-columns: [ads] 200px [main] 1fr [links] 200px;
  grid-template-areas:
    "header  header  header"
    "ads     main    links"
    "footer  footer  footer"
}
```

With the potential for values to become that drawn out and complicated, it's arguable whether the **grid** shorthand is a true shortcut (see **Note**).

There is also a **grid-template** shorthand property that works exactly like **grid**, but it may be used only with explicitly defined grids, as a stand-in for **grid-template rows / grid-template-columns**. The Grid Layout spec strongly recommends that you use the **grid** shorthand instead of **grid-template** unless you specifically want the cascading behavior of **grid-template**.

NOTE

The Grid Layout experts I've talked to don't tend to use **grid** *or* **grid-template** *except for the simplest of grid structures. The code becomes overly complex, and one small slip can make the whole grid fall apart. For complicated grid structures, you'll probably want to stick to separate properties for defining rows, columns, and areas.*

grid Shorthand for Implicit Grids

The **grid** shorthand property can be used with implicit grid properties as well. Adding the **auto-flow** keyword value to either the row or column track information indicates that the tracks on that axis should be automatically generated at the provided dimension.

Say we want to establish columns explicitly, but let rows generate automatically as needed. The implicit grid we made at the beginning of this chapter could be written with the **grid** shorthand like so:

```
#layout {
  display: grid;
  grid: auto-flow 4em / 300px 300px 300px;
}
```

Remember that the **grid** shorthand syntax lists row information first, then a slash, then the column information. Here, the rule says to create rows automatically that are 4em high and create three columns at 300px each. When **auto-flow** is applied to rows, the **grid-auto-flow** property is set to **row**.

When **auto-flow** is applied to the column values, **grid-auto-flow** is set to **column**. Here is how the implicit column example you wrote in the last step of EXERCISE 18-1 could be written with **grid**:

```
dl {
  display: grid;
  grid: repeat(3,3em 1fr) / autoflow 6em;
  gap: .5em;
}
```

It is important to keep in mind that because **grid** is a shorthand property, any omitted value will be reset to its default. Therefore, if you've also used **grid** to set up explicit rows and columns, those will be lost if a **grid** shorthand with implicit grid instructions appears later in the style sheet.

Well, that that was a long explanation for a shorthand property! You now know all the ins and outs of setting up grids and placing items in them. In the next section, you'll get to know the properties for controlling the spacing and alignment of grid cells and items within them. But first, take a look at the **"Grid Item Shuffle"** sidebar to see how the **order** and **z-index** properties you learned about in the previous chapters apply to Grid Layout as well.

GRID ALIGNMENT AND SPACING

Grids use the collection of alignment and spacing properties that "control the alignment of boxes within other boxes," as defined in CSS Box Alignment Module Level 3 (*w3.org/TR/css-align-3/*). The properties were originally extracted from the Flexbox specification, so they should look familiar. The box alignment properties were designed to apply to the block and table display modes as well; however, there is no browser support for those applica-

Grid Item Shuffle

So far, you've seen grid items flowing into a grid sequentially or getting placed into their own little areas explicitly. There are a few properties that may be useful for tweaking the position of grid items.

Changing the order

As with flex items, you can apply the **order** property to a grid item element to change the order in which it appears when it is rendered. Keep in mind that the **order** property does not change the order in which items are read by an assistive device or tabbed through, however, so use it with care. See the section **"Changing the Order of Flex Items"** in **Chapter 17** for more information on how to use **order**. The same principles apply with grid items.

Stacking order

It is possible to position items in a grid in a way that causes them to overlap. When more than one item is assigned to a grid area, items that appear later in the source are rendered on top of items earlier in the source, but you can change the stacking order by using the **z-index** property. Assigning a higher **z-index** value to earlier item elements makes them render above items that appear later. See the section **"Stacking Order"** in **Chapter 16, Floating and Positioning**, for details on using **z-index**.

tions as of this writing. With all of the alignment properties documented in one central specification, there is less chance of conflicts between the various display modes.

Block-axis alignment properties begin with align-.

Inline-axis alignment properties begin with justify-.

The box alignment properties fall into three categories, for:

- Alignment and spacing of boxes within their containers (i.e., tracks within in a grid container)
- Alignment of items within their boxes (i.e., items within their grid cells or areas)
- Spacing added between tracks (the **gap** properties)

The alignment properties use consistent terms for alignment on each axis. The properties that position tracks or items along the block axis (vertically in our examples) begin with **align-**. The properties that align tracks or items along the inline axis (horizontally) begin with **justify-**. Keeping these prefixes in mind can make the set of alignment properties easier to remember.

We're going to start our alignment lesson by moving groups of row and/or column tracks around in their grid container.

Aligning Tracks Within a Grid Container

There may be instances in which the tracks of your grid do not fill the entire area of their grid container—for example, if you've specified three 200-pixel columns in a grid that is 800 pixels wide (see **Note**). You can decide how the browser should handle leftover space within the container by using the **justify-content** (horizontal/inline axis) and **align-content** (vertical/block axis) properties. The **-content** suffix is the Box Alignment module's convention for indicating the alignment of content boxes within their containers.

NOTE

Some track values, such as `fr` *and certain content-based values, allot the extra space to within the tracks themselves to fill the available space in the grid, so the alignment properties will not have an effect.*

justify-content

Values:	`start` | `end` | `left` | `right` | `center` | `stretch` | `space-around` | `space-between` | `space-evenly`
Default:	`start`
Applies to:	Grid containers
Inherits:	No

NOTE

The `stretch` *keyword works only when the track width or height is set to* `auto`.

align-content

Values:	`start` | `end` | `left` | `right` | `center` | `stretch` | `space-around` | `space-between` | `space-evenly`
Default:	`start`
Applies to:	Grid containers
Inherits:	No

In FIGURE 18-21, the grid container is indicated with a gray outline. The rows and columns of the drawn grid do not fill the whole container, so something has to happen to that extra space. The **start**, **end**, and **center** keywords

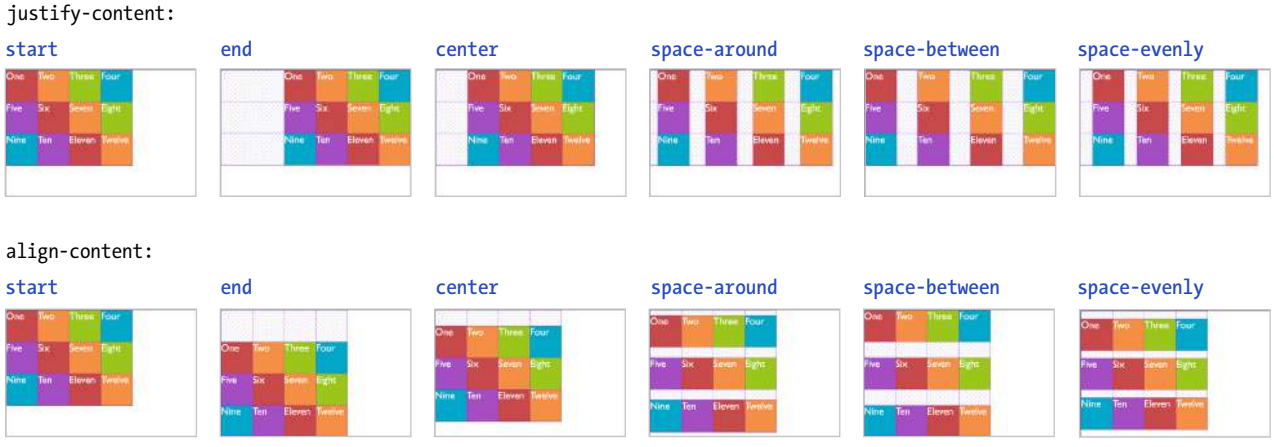


FIGURE 18-21. The `justify-content` and `align-content` properties distribute extra space in the container.

move all the tracks around within the container by putting the extra space after, before, or equally on either side of them, respectively. The **`space-around`** and **`space-between`** keywords distribute space around tracks, as discussed in **Chapter 17**. The **`space-evenly`** keyword adds an equal amount of space between items and at the start and end of each track.

NOTE

When you distribute space around and between tracks, it adds to whatever `gap` *settings you may have.*

Aligning Items Within Their Grid Areas

Similarly, you can align items within their grid areas when there is leftover space. We'll start by looking at the properties for adjusting the alignment of individual items and follow up with properties that change the alignment of all grid areas at once.

Aligning individual items

`justify-self`

Values: `start` | `end` | `center` | `left` | `right` | `self-start` | `self-end` | `stretch` | `normal` | `auto`

Default: `auto` (looks at the value for `justify-items`, which defaults to `normal`)

Applies to: Grid items

Inherits: No

`align-self`

Values: `start` | `end` | `center` | `left` | `right` | `self-start` | `self-end` | `stretch` | `normal` | `auto`

Default: `auto` (looks at the value for `align-items`)

Applies to: Grid items

Inherits: No

When a grid item doesn't fill its entire grid area, you can specify how you'd like it to be aligned within that space. As you would expect, the **justify-self** property aligns on the inline axis and **align-self** specifies alignment on the block axis. These properties apply to the grid item element, which makes sense because you want that item to align itself.

FIGURE 18-22 shows the effects of each keyword value. For items with their size set to **auto** (or in other words, not explicitly set with **width** and **height** properties), the default is **stretch**. This is what you've seen in all the previous grid examples. If the grid item has a width and height specified, those dimensions are preserved and the default is **start**.

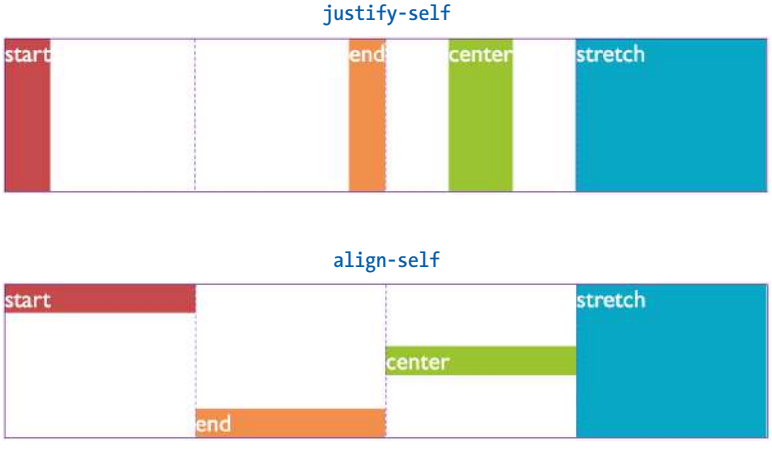


FIGURE 18-22. Values for **justify-self** and **align-self** for aligning a single grid item within its respective grid area. These values have the same use in the **justify-items** and **align-items** properties that are used to align all the items in the grid.

NOTE

The **self-start** *and* **self-end** *values look at the writing direction of the content of the item and use its start or end edge for alignment. For example, if an item is in Arabic, its* **self-start** *edge is on the right, and it will be aligned to the right. The* **start** *and* **end** *values consider the writing direction of the grid container. The* **left** *and* **right** *keywords are absolute and will not change with the writing system, but they correspond to* **start** *and* **end** *in left-to-right languages.*

■ TIP

If you want a grid item to stay centered in its grid area, set both **align-self** and **justify-self** to **center**.

Aligning all items in their grid areas

justify-items

Values: start | end | center | left | right | self-start | self-end | stretch | normal

Default: normal (stretch for non-replaced elements; start for replaced elements)

Applies to: Grid containers

Inherits: No

align-items

Values: start | end | center | left | right | self-start | self-end | stretch | normal

Default: normal (stretch for non-replaced elements; start for replaced elements)

Applies to: Grid containers

Inherits: No

To align all of the items in a grid in one fell swoop, use the **justify-items** property for inline axis alignment and **align-items** for the block axis. Apply these properties to the grid container element so it affects all of the items in the grid. The keywords do the same things shown in FIGURE 18-22; just picture it happening consistently across the entire grid. Keep in mind that these settings will be overridden by the **-self** properties.

Adding Space Between Tracks (gap)

For some grids, it's nice to have a little breathing room between rows or columns (or both!) without needing to set up extra tracks just to hold space. Adding gutters between tracks is the job of the collection of **gap** properties (see **Note**) defined in the Box Alignment module. You may remember that **gap** can also be used with flexboxes and multicolumn layouts.

row-gap
column-gap

Values: *length (must not be negative)*

Default: 0

Applies to: Grid containers

Inherits: No

gap

Values: *grid-row-gap grid-column-gap*

Default: 0 0

Applies to: Grid containers

Inherits: No

You've gotten a chance to use the **gap** property in the exercises, so this should sound familiar. Setting a length value for **row-gap** adds space between the row tracks of the grid, and **column-gap** adds space between column tracks. The effect is as if the grid lines have a width; however, the gap width applies only to lines between tracks, not to the first and last lines on the sides of the grid.

You can use the **gap** shorthand to specify gap widths for rows and columns in one go, with rows first as usual, separated by a character space. In this example, I've added 20px of space between rows and 50px of space between columns by using the **gap** shorthand (FIGURE 18-23):

```
#container {
  border: 2px solid gray;
  display: grid;
  grid: repeat(4, 150px) / repeat(4, 1fr);
  gap: 20px 50px;
 }
```

Speaking of Spacing, What About Margins?

You can add margins to a grid item as you can for any other element. It is useful to know that the item's margin box will be anchored to the cell or grid area, and the margin space is preserved.

You can use margins to move the item around in the grid area. For example, setting the left margin to **auto** pushes the item to the right, as we saw in the Flexbox examples in the previous chapter. Setting the left and right margins to **auto** (as long as item has a specified width) centers it horizontally. In Grid Layout, you can also set the top and bottom margins to **auto** and, as long as there's a specified height, the item will be centered vertically. Of course, you have the box alignment properties to achieve these effects as well.

NOTE

In earlier versions of the Grid Layout spec, these properties were prefixed with **grid-** *(for example,* **grid-row-gap***). The unprefixed versions from the Box Alignment module are well supported in modern browsers; however, browsers will continue to support the prefixed versions for backward compatibility.*

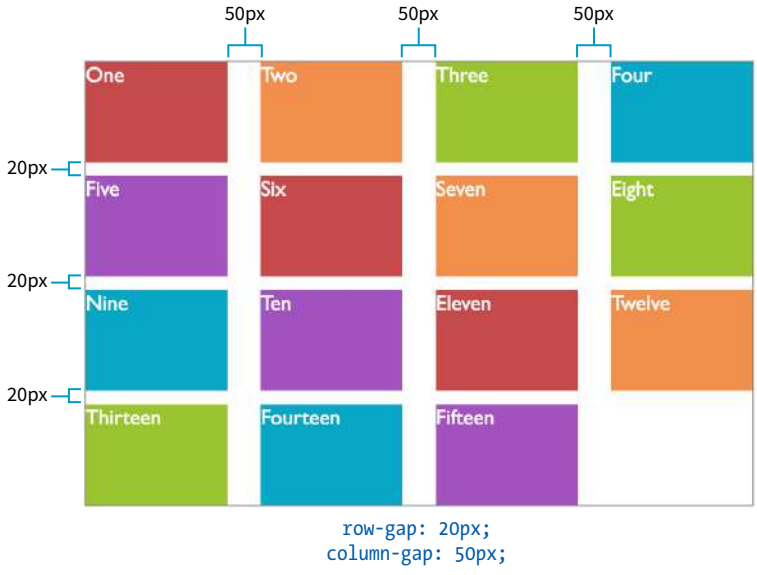


FIGURE 18-23. Grid gaps add gutter spaces between tracks.

Because the best way to get a feel for how the box alignment properties work is to play around with them yourself, EXERCISE 18-4 gives you a chance to do exactly that.

RESPONSIVE GRIDS

Although my deeper explanation of responsive web design is waiting for you in **Chapter 19, Responsive Web Design**, you should already be familiar with the term. As a refresher, responsive web design is a strategy for creating sites that change or adapt based on the size of the viewport. In this way, the same HTML document may be arranged differently based on whether it is displayed on a smartphone, tablet, or large screen.

You may tend to think of a grid as being a rigid structure, but there are several methods for incorporating Grid Layout into responsive layouts.

The fr Unit

The easiest way to introduce some flexibility into your grid is to use `fr` (fractional) units for column widths. They won't solve all design challenges, but they are a good starting place to make sure your tracks shrink or grow to fill the available space. I discussed `fr` units in the **"Specifying Track Size Values"** section earlier in this chapter, but here is a quick example that specifies four columns that will grow or shrink as space allows, with the fourth column always four times the width of the other three:

```
grid-template-columns: 1fr 1fr 1fr 4fr;
```

EXERCISE 18-4. Playing with alignment

I've provided the source document (*grid-alignment.html*) used for FIGURE 18-21 with the materials for this chapter. Unlike other exercises, this one is about experimenting without a specific goal in mind. I'll provide a few steps you can follow as suggestions, but the idea here is for you to test out different alignment combinations and see the results. Some may surprise you.

1. Open *grid-alignment.html* in a browser. You will see the default alignment of the items and tracks, because no alignment properties have been applied yet. Use the Grid Inspector in your browser's Developer's Tools (I'm using Chrome) to show the structure of the grid. This will help you visualize the results of the alignment values. We'll start by manipulating the alignment of the tracks within the grid container with the **-content** properties.
2. Open the document in a text or code editor, and let's begin adding some styles. Find the **align-content** declaration and remove the comment indicators (**/* */**) to make it active. Start by changing the value to **center**; save and reload the page in the browser and notice the new position. Now try **space-between**. Set it back to the **start** value for the next step.
3. Uncomment the **justify-content** declaration. Try out the **end** and **space-evenly** values, saving and viewing the document in the browser after each step. The full list of values for **align-content** and **justify-content** is:

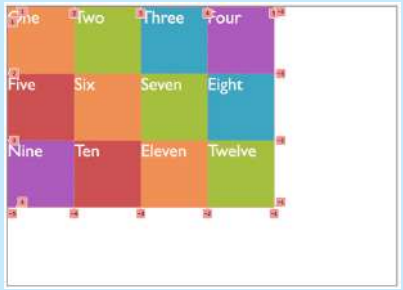
   ```
   start | end | left | right | center | stretch |
   space-around | space-between | space-evenly
   ```

   Try out various combinations of values for the two track-alignment properties to see how they interact. When you are finished, leave them both at **start** so you have a neutral starting point for the next steps.
4. Now let's see what the **align-items** and **justify-items** properties do. I've applied colored backgrounds to the item elements so their boundaries are clear. Because the default value for the **-items** properties is **stretch**, the items (and their backgrounds) fill their cells. But that changes with other values, as you will see. Start by uncommenting the **align-items** property to adjust the vertical alignment of the items within their cells. Set the value to **end**. Save and view it in the browser. The background colors indicate that the item element boxes stretch across their columns but are just high enough to contain their text content. Try the **center** value to center the items vertically.
5. Do the same thing for the **justify-items** property. Try the **end** property. Change it to **center**. Play around with combinations of values for **align-items** and **justify-items**:

   ```
   start | end | center | left | right | self-start |
   self-end | stretch | normal
   ```

   For additional practice, you can try placing items into grid areas that span over several cells by using the item placement properties (**grid-row-start**, **grid-row-end**, etc.). How do the track and item alignment values behave when the grid is divided up into unequal areas?
6. When you're done experimenting with alignment values, hide all of the alignment properties in a comment, which is one way to set all of the values back to their defaults. With a clean slate, add space between tracks with **column-gap** and **row-gap** lengths. Try combining the gaps with track spacing and see what happens.
7. If you'd like to play with the alignment of a grid that has images instead of text, I've also provided the document for the photo gallery example (*gallery-alignment.html*). See if you can achieve the alignment you think looks best for the content.

Default alignment:




```
align-content: start;
justify-content: start;

align-items: stretch;
justify-items: stretch;
```

```
align-content: center;
justify-content: center;

align-items: center;
justify-items: center;
```

FIGURE 18-24. The Grid Inspector reveals the grid structure with default alignment and how it displays when all of the alignment properties are set to **center**.

minmax()

Setting track sizes with the **minmax()** function also introduces flexibility, but it allows authors to set limits on how narrow or how wide each track should grow. Here, I've specified that the last column may not be any smaller than 24em but it can flex as wide as space allows:

```
grid-template-columns: 1fr 1fr 1fr minmax(24em 1fr);
```

auto-fill and auto-fit

When I introduced track size values, we looked at the **repeat()** function for specifying repeating patterns of track sizes. In this example, the **repeat()** function is used to create exactly six columns by repeating 20em and 40em track widths three times:

```
grid-template-columns: repeat(3, 20em 40em);
```

Another way to use **repeat()** is to specify one track size and let the browser figure out how many tracks to repeat based on the available space. This is done by using the **auto-fit** or **auto-fill** value instead of a repeat-count integer (3, in the previous example). These values are great for implicit grids where you want to take advantage of the vertical and horizontal alignment of a grid, but are not tied to a particular number of columns.

Let's see how this works using our simple five-item example from earlier in the chapter. I know I'd like every column to be 200 pixels wide, but I don't know how many columns will fit when it arrives on the user's device. The **auto-fit** value tells the browser "make as many tracks as will fit until all items are placed." The value after the comma sets the track width:

```
grid-template-columns: repeat(auto-fit, 200px);
```

FIGURE 18-25 shows the result when the grid is displayed in wide and narrow browser windows. Notice that in the wide window, the grid is five columns wide with extra space on the right; this is because there are only five items, and **auto-fit** stops making tracks when it runs out of items.

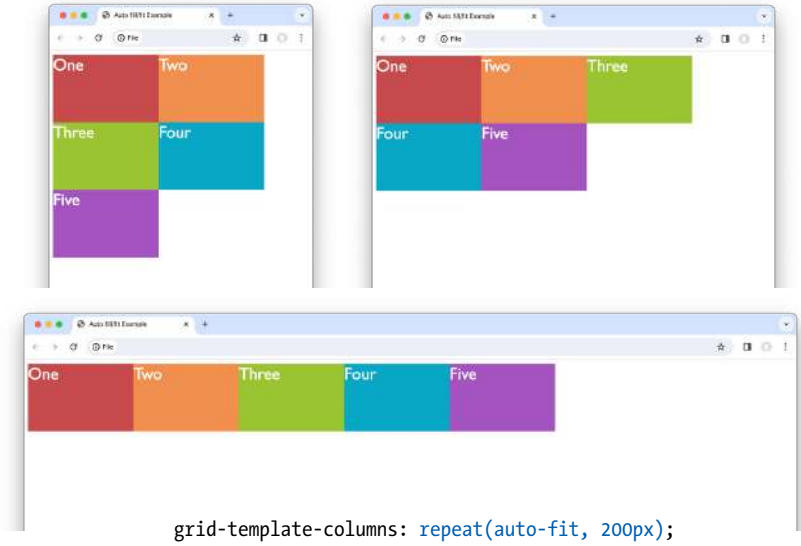
The **auto-fill** value works similarly, except that it creates as many tracks as will fit, independent of the number of items. In FIGURE 18-26, the Grid Inspector reveals that the **auto-fill** grid has seven columns, with two cells held empty. It's a subtle difference, and although **auto-fit** is more commonly used, both have their uses.

It's worth noting that you can use **auto-fit** and **auto-fill** for creating row tracks based on the available height of the grid as well; however, in the context of responsive web design, width is generally the constraining dimension.

I've only scratched the surface of working with **auto-fit** and **auto-fill** here. For a more in-depth explanation, I recommend the article "Auto-Sizing Columns in CSS Grid: `auto-fill` vs `auto-fit`" by Sara Soueidan at CSS-Tricks (*oreil.ly/NkaOb*).

WARNING

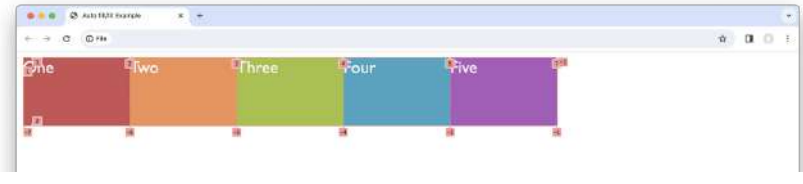
*Content-based keywords (***auto***,* **min-content***, and* **max-content***) cannot be used with* **auto-fill** *or* **auto-fit***.*




```
grid-template-columns: repeat(auto-fit, 200px);
grid-auto-rows: 4em;
```

FIGURE 18-25. The `auto-fit` value for `repeat()` adds as many tracks as will fit in the browser window until all items are placed.

```
repeat(auto-fit, 200px);
grid-auto-rows: 4em;
```




```
repeat(auto-fill, 200px);
grid-auto-rows: 4em;
```

FIGURE 18-26. `auto-fit` makes just enough tracks to contain the number of items in the grid. `auto-fill` fills the entire space, even if some cells are empty.

Media Queries

The `auto-fit`/`auto-fill` approach works great for implicit grids, but if you are using an explicit grid to lay out page components deliberately, that may not be the behavior you want. For instances when you need a completely different grid structure on small, medium, and large screens, the solution is to deliver an appropriate grid with a media query. A **media query** checks for aspects of the viewing environment (such as viewport width) and serves a set of styles only when a specified requirement is met. For example, if the browser detects that the viewport is only 600 pixels wide, it will grab the styles for the grid you designed to work best at that size.

That's all I'm going to say about media queries right now, because I'll cover them in depth in the next chapter. For now, keep in mind that it is possible to create a few different grid layouts aimed at various device widths and get them to the right users. You can combine this approach with values that give grids flexibility for the in-between sizes as well. For example, you may use `fr` or `minmax()` values to make sure two columns fill the screen on a variety of tablets, but switch to a three-grid design when the page is viewed on a large desktop monitor.

Before we wrap up the chapter, let's convert the Black Goose Bakery site we've worked on in previous chapters to a grid layout in EXERCISE 18-5. This will be the first step toward making it a fully responsive site in **Chapter 19**.

EXERCISE 18-5. A grid layout for the bakery page

The Black Goose Bakery page has come a long way. You've added padding, borders, and margins. You've floated images, positioned an award graphic, and created a navigation bar using Flexbox. Now you can use your new Grid Layout skills to give the page a two-column layout that will be appropriate for tablets and larger screens (FIGURE 18-27).

Start by opening the *bakery.html* and *bakery-styles.css* files as you left them in EXERCISE 17-1. If you'd like a fresh start, you will also find starter documents with the materials for this chapter.

1. We need to add a bit of markup that encloses the content in an element to serve as the grid container. In *bakery.html*, add a `div` around all of the content elements (from `header` to `footer`) and give it the `id` "container" (note that we could have also used the `body` element as there is no content outside the container `div` in this example). Save the HTML file.

```
<body>
  <div id="container">
    <header>...</header>
    <main>...</main>
    <aside>...</aside>
    <footer>...</footer>
  </div>
</body>
```

EXERCISE 18–5. Continued

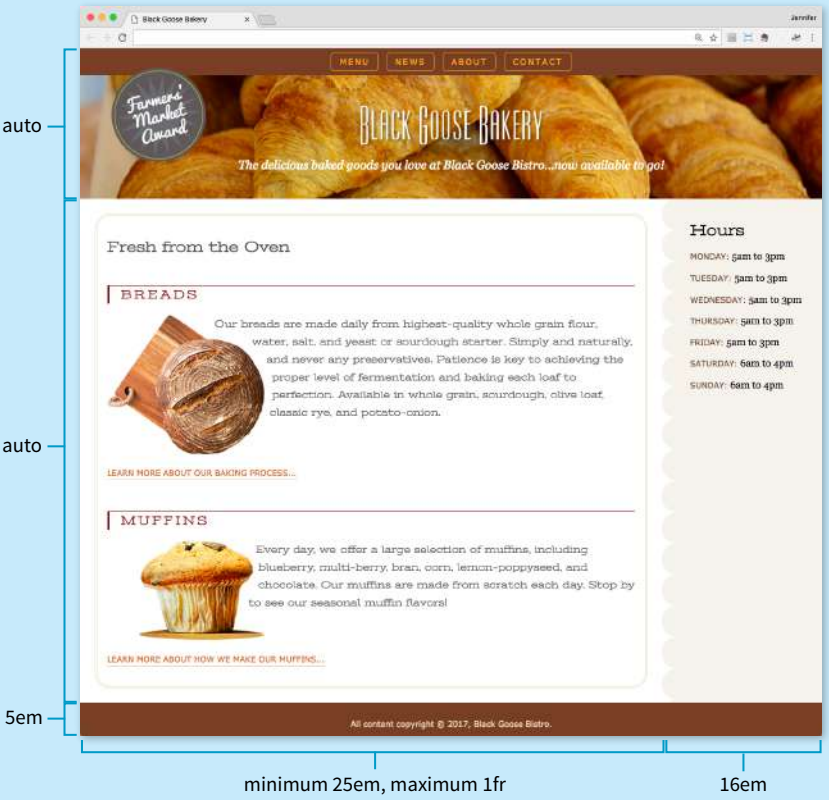

auto
auto
5em
MENU
NEWS
ABOUT
CONTACT
Farmers' Market Award
BLACK GOOSE BAKERY
The delicious baked goods you love at Black Goose Bistro...now available to go!
Fresh from the Oven
BREADS
Our breads are made daily from highest-quality whole grain flour, water, salt, and yeast or sourdough starter. Simply and naturally, and never any preservatives. Patience is key to achieving the proper level of fermentation and baking each loaf to perfection. Available in whole grain, sourdough, olive loaf, classic rye, and potato-onion.
LEARN MORE ABOUT OUR BAKING PROCESS...
MUFFINS
Every day, we offer a large selection of muffins, including blueberry, multi-berry, bran, corn, lemon-poppyseed, and chocolate. Our muffins are made from scratch each day. Stop by to see our seasonal muffin flavors!
LEARN MORE ABOUT HOW WE MAKE OUR MUFFINS...
Hours
MONDAY: 5am to 3pm
TUESDAY: 5am to 3pm
WEDNESDAY: 5am to 3pm
THURSDAY: 5am to 3pm
FRIDAY: 5am to 3pm
SATURDAY: 6am to 4pm
SUNDAY: 6am to 4pm
All content copyright © 2017, Black Goose Bistro.
minimum 25em, maximum 1fr
16em

FIGURE 18-27. The Black Goose Bakery page with a two-column grid layout

In the style sheet (*bakery-styles.css*), add a style rule to make the new **div** display as a grid:

```
#container {
  display: grid;
}
```

2. First we'll work on the rows. FIGURE 18-27 shows that we need three rows to create the layout. Set the height of the first row track to **auto** so it will observe the height settings on the elements within it and automatically accommodate the content. The second row has a lot of text, so use the **auto** track value again to guarantee the track will expand at least as much as necessary to fit the text. For the third row, a height of 5em should be sufficient to fit the few lines of text with a comfortable amount of space:

```
#container {
  display: grid;
  grid-template-rows: auto auto 5em;
}
```

3. Now we can set up the column tracks. It looks like we'll need only two: one for the main content and one for the Hours sidebar. We've used the **minmax()** function so we can ensure the text column never gets narrower than 25em but can expand to fill the available space in the browser (1fr). The Hours column feels right to me at 16em, but feel free to try other values:

EXERCISE 18–5. Continued

```
#container {
  display: grid;
  grid-template-rows: auto auto 5em;
  grid-template-columns: minmax(25em, 1fr) 16em;
}
```

4. Next, let's name the areas in the grid so we can place the items in them easily and efficiently. Use the **grid-template-areas** property to name the cells in the grid:

```
#container {
  display: grid;
  grid-template-rows: auto auto 5em;
  grid-template-columns: minmax(25em, 1fr) 16em;
  grid-template-areas:
    "banner banner"
    "main   hours"
    "footer footer";
}
```

5. With everything set up, it'll be a breeze to put the content items into their proper places. Create a style rule for each grid item and tell it where to go with **grid-area**:

```
header {
  grid-area: banner;
}
main {
  grid-area: main;
}
aside {
  grid-area: hours;
}
footer {
  grid-area: footer;
}
```

Pretty easy, right? Now would be a good time to save the file and take a look at it in the browser (if you haven't already). It's looking good (hopefully it is for you too), but let's tweak the margins to make them more appropriate for the new layout. Change the **main** margin (in the **main** style declaration lower in the style sheet) from 2.5% to 20px so it doesn't react to the width of the viewport window.

```
main {
  ...
  margin: 2.5% 20px;
```

Next, remove the margin and border radius I had set on the **aside** so it fills the right column. You can just comment them out so that information is still around if you need to use it later:

```
aside {
  ...
  /* border-top-right-radius: 25px;
  margin: 1em 2.5% 0 75px; */
}
```

That does it! Open the *bakery.html* page in a browser, and it should look like the screenshot in FIGURE 18–27.

Now the bakery page has a nice two-column layout using a simple grid. In the real world, this would be just one layout in a set that would address different screen sizes as part of a responsive design strategy.

ADDITIONAL RESOURCES

As you continue your Grid Layout adventure, I'm sure you'll find plenty of excellent resources online, but this is a good list to get you started.

CSS: The Definitive Guide, 5th edition, by Eric A. Meyer **and Estelle Weyl (O'Reilly)**

Do you like big books? *CSS: The Definitive Guide* has 1,126 pages covering everything you want to know about CSS, including an entire chapter devoted to Grid Layout. Eric and Estelle have a knack for including every last detail yet making it all understandable.

Mastering CSS Grid: A Comprehensive and Practical Guide to Creating Beautiful Layouts with CSS Grid by Pascal Thormeier **(Packt)**

This is a deep dive into using Grid Layout and Flexbox to create complex layouts that covers advanced topics as well.

CSS Grid Layout Module Level 2 *(w3.org/TR/css-grid-2/)*

The spec itself is the best source for every last detail and the last word on Grid Layout.

Rachel Andrew's "Grid by Example" site *(gridbyexample.com)*

Rachel Andrew, one of the first champions of Grid Layout, has assembled an incredible collection of articles, free video tutorials, browser support information, and more. You can also try searching the web for her excellent conference talks on the topic.

Jen Simmons's "Experimental Layout Lab" *(labs.jensimmons.com)*

Jen Simmons, also known for her work promoting grids, shows off what Grid Layout can do in her Experimental Layout Lab. It's definitely worth a visit for the cool examples of Grid Layout and other emerging CSS technologies, as well as exercises that let you code along.

MDN Web Docs *(oreil.ly/bh7BD)*

As for all web code–related topics, there is a lot of great Grid Layout information on the MDN Web Docs site. If the W3C spec feels too dense to read (as it sometimes does to your humble author), MDN Web Docs can help you make sense of it. See also the guides with a more instructional tone: *oreil.ly/_uDUfs*.

Grid Garden, by Thomas Park *(cssgridgarden.com)*

If you enjoyed the Flexbox Froggy game by Thomas Park, you will have fun playing his Grid Garden game for getting familiar with Grid Layout.

TEST YOURSELF

We covered a lot of ground in this chapter. See how you do on this quiz of some of the highlights. As always, **Appendix A** has the answers.

1. What is the key difference between Grid Layout and Flexbox? Name at least one similarity (there are multiple answers).

2. Create the grid template for the layout shown in FIGURE 18-28 by using `grid-template-rows` and `grid-template-columns`. Write it again, this time using the `grid` shorthand property.

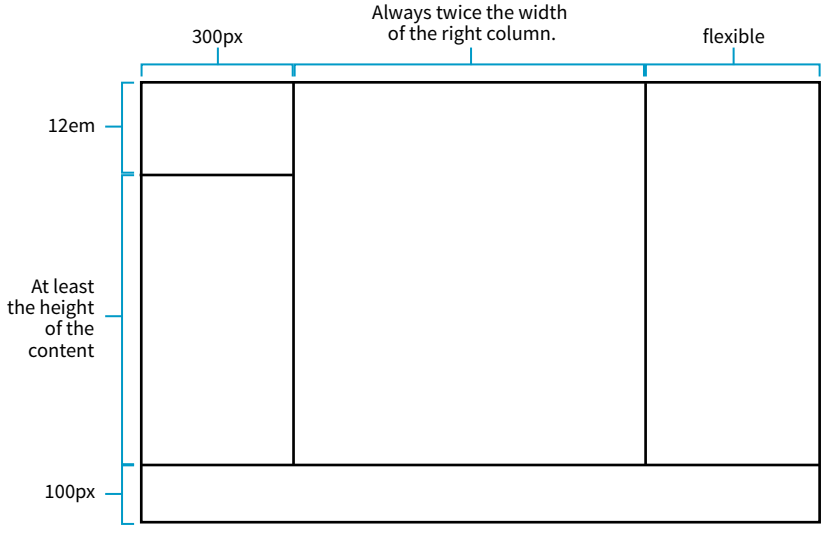


FIGURE 18-28. Create the grid template for this grid structure.

3. Match the following style declarations with the lettered grid items in FIGURE 18-29. In addition to automatic numbering, some of the grid lines have been named, as labeled.

   a.
   ```
   grid-row-start: 1;
   grid-row-end: 3;
   rid-column-start: 3;
   grid-column-end: 7;
   ```

   b.
   ```
   grid-area: 2 / 2 / span 4 / 3;
   ```

   c.
   ```
   grid-area: bowie;
   ```

   d.
   ```
   grid-row: -2 / -1;
   grid-column: -2 / -1;
   ```

e.
```
grid-row-start: george;
grid-row-end: ringo;
grid-column-start: paul;
grid-column-end: john;
```

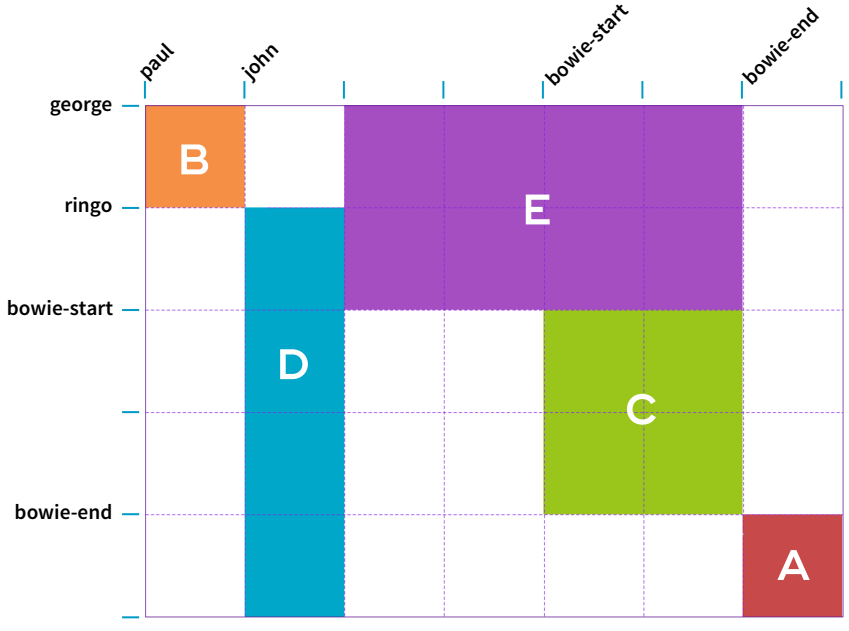


FIGURE 18-29. Match the style examples to the items in this grid.

4. Write a style rule that adds 1em of space between columns in a grid container named `#gallery`.

5. Match the tasks with the declarations.

a. _____ `justify-self: end;`

b. _____ `align-items: end;`

c. _____ `align-content: center;`

d. _____ `align-self: stretch;`

e. _____ `justify-items: center;`

1. Make a particular item stretch to fill its container.
2. Position an image on the right edge of its grid area (in a left-to-right reading language).
3. Center the whole grid vertically in its container.
4. Push all of the images in a grid to the bottom of their respective cells.
5. Center all items in their areas horizontally.

CSS REVIEW: LAYOUT PROPERTIES

Here are the properties covered in this chapter, sorted by whether they apply to the container or items.

Grid container properties	
`display: grid | inline-grid`	Sets the display mode of an element to a grid context
`grid-template`	Shorthand property for specifying **`grid-template-areas`**, **`grid-template-rows`**, and **`grid-template-columns`**
`grid-template-areas`	Assigns names to areas in the grid
`grid-template-columns`	Specifies track sizes for the columns in explicit grids
`grid-template-rows`	Specifies track sizes for the rows in explicit grids
`grid-auto-columns`	Specifies track sizes for automatically generated columns
`grid-auto-flow`	Indicates the direction and density in which items flow automatically into a grid
`grid-auto-rows`	Specifies track sizes for automatically generated rows
`grid`	Shorthand property for specifying **`grid-template-rows`**, **`grid-template-columns`**, and **`grid-template-areas`** or **`grid-auto-flow`**, **`grid-auto-rows`**, and **`grid-auto-columns`**
`gap`	Shorthand property for specifying **`row-gap`** and **`column-gap`**
`column-gap`	Specifies the width of the gutter between columns
`row-gap`	Specifies the width of the gutter between rows
`justify-items`	Indicates alignment of all the grid items along the inline axis within their respective areas
`justify-content`	Indicates alignment of the grid tracks along the inline axis in the container
`align-items`	Indicates alignment of all the items in a grid along the block axis within their respective grid areas
`align-content`	Indicates alignment of the grid tracks along the block axis in the container
Grid item properties	
`grid-column`	Shorthand property for specifying **`grid-column-start`** and **`grid-column-end`**
`grid-column-end`	Denotes the end line of the column in which an item is to be placed
`grid-column-start`	Denotes the start line of the column in which an item is to be placed
`grid-row`	Shorthand property for specifying **`grid-row-start`** and **`grid-row-end`**
`grid-row-end`	Denotes the end line of the row in which an item is to be placed
`grid-row-start`	Denotes the start line of the row in which an item is to be placed
`grid-area`	Assigns a grid item to a named area or an area described by its four boundary grid lines
`align-self`	Indicates alignment of a single item along the block axis within its grid area
`justify-self`	Indicates alignment of a single grid item along the inline axis within its area
`order`	Specifies the order in which to display the item relative to other items in the source
`z-index`	Specifies the stacking order of an item relative to other items when there is overlap

CHAPTER
19

RESPONSIVE WEB DESIGN

IN THIS CHAPTER

What RWD is and why it's important

Fluid layouts

Media queries

Container queries

Design strategies and patterns

Testing options

I first introduced you to the concept of responsive web design way back in **Chapter 3, Some Big Concepts You Need to Know**, and I've been addressing ways to keep all screen sizes in mind throughout this book. In this chapter, I'll go into more detail on responsive strategies and techniques.

Just to recap, responsive web design (RWD) is a design and production approach that allows a website to be comfortably viewed and used on all manner of devices. The core principle is that all devices get the same HTML source, located at the same URL, but different styles are applied based on the viewport size and other viewing environment factors to rearrange components and optimize usability. FIGURE 19-1 shows examples of responsive sites as they might appear on a smartphone, tablet, and desktop, but it is important to keep in mind that these sites are designed to work well on the continuum of every screen width in between.

WHY RWD?

Since the iPhone shook things up in 2007, folks now view the web on phones of all sizes, tablets, touch-enabled laptops, headset computers, televisions, video game consoles, refrigerators, and who knows what else that may be coming down the line.

In 2016, mobile internet usage surpassed desktop usage—an important milestone. For roughly 15% of Americans, a smartphone or tablet is their *only* access to the internet because of lack of access to a computer or high-speed WiFi at work or home,* and the percentage of web traffic that comes from

* Risa Gelles-Watnick, "Americans' Use of Mobile Technology and Home Broadband," Pew Research Center, January 31, 2024, *pewresearch.org/internet/2024/01/31/americans-use-of-mobile-technology-and-home-broadband/*.

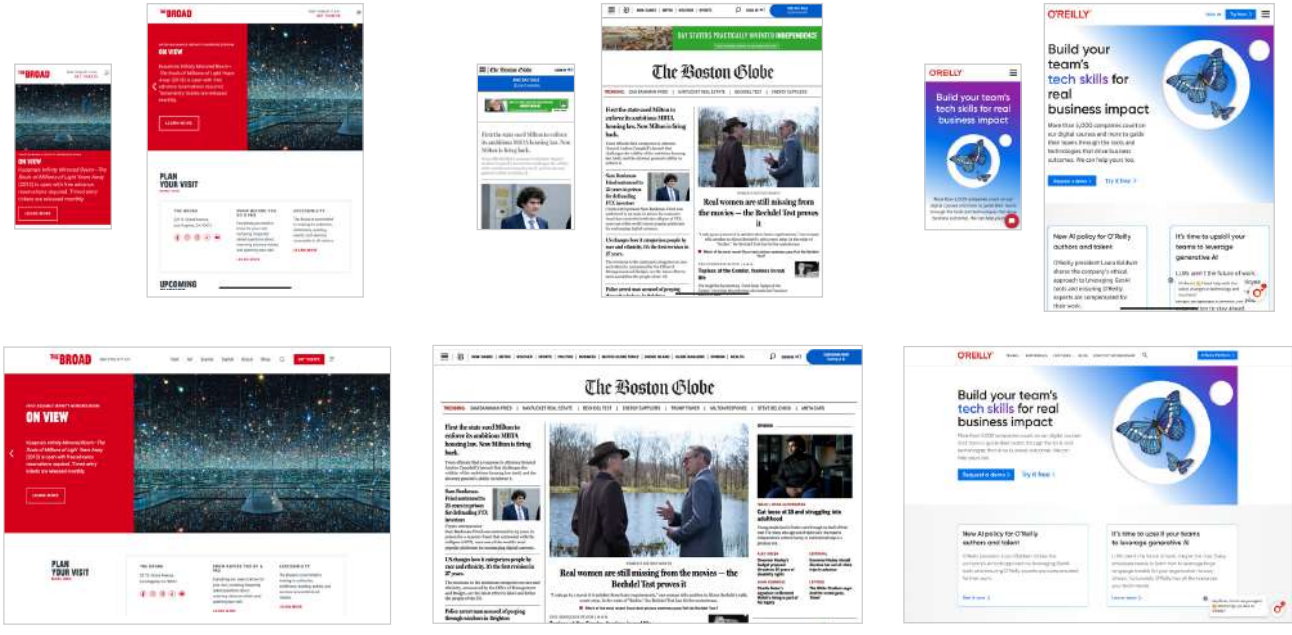


FIGURE 19-1. Examples of responsive sites that adapt to fit small, medium, and large screens and all sizes in between

NOTE

Each site in FIGURE 19-1 *has one morphing design, not three distinct layouts. Some sites do have separate static layouts aimed at specific devices, which is an approach known as* **adaptive design**.

devices other than desktop browsers is steadily increasing. Younger users may be mobile-only by choice. Furthermore, the majority of us access the web from a number of platforms (phone, tablet, computer) over the course of the day. And guess what—we expect to have a similar experience using your content or service regardless of how we access your site.

Responsive design is the default way to build websites that meet the demands of our current multidevice environment.

That's where RWD fits in. With one source, you ensure that mobile visitors receive the same content as other visitors (although it might be organized differently). Users are not penalized with stripped-down content or features just because they are using a smartphone. And for visitors who might start using your site on one device and then switch to another, you can ensure a consistent experience.

In fact, these days "responsive design" is now just "web design." Instead of a niche approach, it has become the default way to build a website that meets the demands of our multidevice environment.

THE RESPONSIVE RECIPE

The deluge of web-enabled mobile devices initially sent shockwaves through the web design community. Accustomed to designing exclusively for large desktop screens, we were unclear on how to accommodate screens that fit in the palm of your hand.

One option was to rely on the phone's built-in web display functionality. By default, mobile devices display an entire web page shrunken down to fit on whatever screen real estate is available. Users can pinch to zoom into the details and scroll around to various parts of the page. While that technically works, it is far from an optimal experience. Another approach in the past was to create a separate mobile site just for small screens and people "on the go." There are still a few dedicated mobile ("m-dot") sites out there—Facebook comes to mind—but in general, m-dot sites have faded away in favor of RWD. Google helped the process along by favoring responsive sites with single URLs over *m.* or *mobile.* versions in search results.

Back in 2010, Ethan Marcotte gave name to a more flexible solution, which has since become a cornerstone of modern web design, in his article "Responsive Web Design" (***alistapart.com/article/responsive-web-design***). In this chapter, I will follow the general "ingredients" for RWD that Ethan outlines in his book *Responsive Web Design*, 2nd ed. (read it for free at ***ethanmarcotte.com/books/responsive-web-design***).

The original responsive design technique has three components:

A flexible layout

: Rather than remaining at a static width, responsive sites use methods that allow them to squeeze and flow into the available browser space.

Flexible images

: Images and other embedded media need to be able to scale to fit their containing elements.

CSS media queries

: Media queries give us a way to deliver sets of style rules only to devices that meet certain criteria, such as screen width and orientation.

To Ethan's ingredient list, I would add the viewport `meta` element, which makes the width of the web page match the available space on the screen or in the browser window. That's where we'll begin our tour of the mechanics of RWD.

SETTING THE VIEWPORT

To fit standard websites onto small screens, mobile browsers render the page on a canvas called the **viewport** and then shrink that viewport down to fit the width of the screen (**device width**). For example, on iPhones, mobile

Safari sets the default viewport width to 980 points (see **Note**), so a web page is rendered as though it were on a desktop browser window set to 980 pixels wide. That rendering gets shrunk down to the width of the screen (ranging from 375 to 430 points, depending on the phone model), cramming a lot of information into a tiny space. Most browsers on Android use a default viewport width of around 980 pixels as well.

Mobile Safari introduced the viewport `meta` element, which allows us to define the size of that initial viewport and override the 980-pixel default. Soon, the other mobile browsers followed suit. The following `meta` element, which goes in the `head` of the HTML document, tells the browser to set the width of the viewport equal to the width of the device screen (`width=device-width`), whatever that happens to be (FIGURE 19-2). The `initial-scale` value sets the zoom level to 1 (100%):

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Including the viewport `meta` element in your HTML document ensures that if the device's screen is 400 points wide, the browser's viewport on that device will also be 400 points across (not 980) and the page will appear on the screen at full size. That is the width we test for with media queries, so setting the viewport is a crucial first step.

NOTE

iOS layouts are measured in points (pt), a unit of measurement that is independent of the number of pixels that make up the physical screen. Points and device pixels are discussed in more detail in **Chapter 26, Web Image Basics**.

WARNING

The viewport `meta` *element also accepts the* `maximum-scale` *attribute. Setting it to 1 (*`maximum-scale=1`*) prevents users from zooming the page, but it is strongly recommended that you avoid doing so because resizing is important for accessibility and usability.*

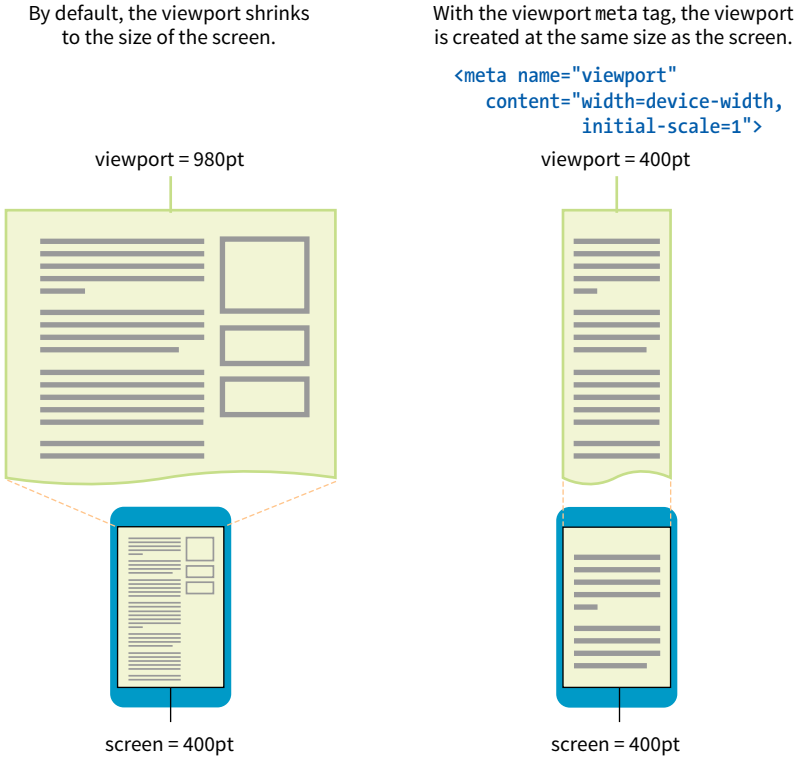


FIGURE 19-2. The viewport `meta` element matches the resolution of the device's browser viewport to the resolution of its screen.

FLUID LAYOUTS

Fluid layouts (or "flexible grids," as Ethan Marcotte calls them in his article and book) are the foundation of responsive design. In 2010, when Ethan coined the term, the only tools we had for achieving flexibility were the normal flow and "columns" made with floats and positioned elements specified with percentage values. In fact, the method for converting pixel measurements to percentages was a key takeaway from his book.

These days, we have more sophisticated tools for making our layouts and the components within them flexible. In 2019, Jen Simmons introduced the term **intrinsic design** to describe the new methodology of designing flexible, multicolumn layouts using the latest CSS features like Flexbox, Grid Layout, and more (see **Note**). The intrinsic design approach opens up layout possibilities beyond the simple recipe Ethan outlined, while still adhering to the basic foundation of a single HTML file serving all devices.

NOTE

You can watch Jen's 2019 presentation at the An Event Apart conference on YouTube: oreil.ly/LevHr.

In a **fluid layout**, the page area and its grid resize to fill the available width of the screen or window (FIGURE 19-3, top). That is easily accomplished with `fr` and `minmax()` units in Grid Layout and with `flex` property settings in Flexbox. Additionally, percentage values for horizontal measurements (so elements remain proportional at varying sizes) are still appropriate for responsive layouts.

By contrast, **fixed-width layouts** are created at a specific pixel width, with 960 pixels being quite fashionable back in the day. Before smartphones came along, when we knew that everyone used our sites on desktop monitors, fixed-width layouts were the norm. (Ahh, those "simple" pre-mobile days when we only needed to deal with radically incompatible browser support!) Specifying all measurements in pixel values gave designers the same kind of

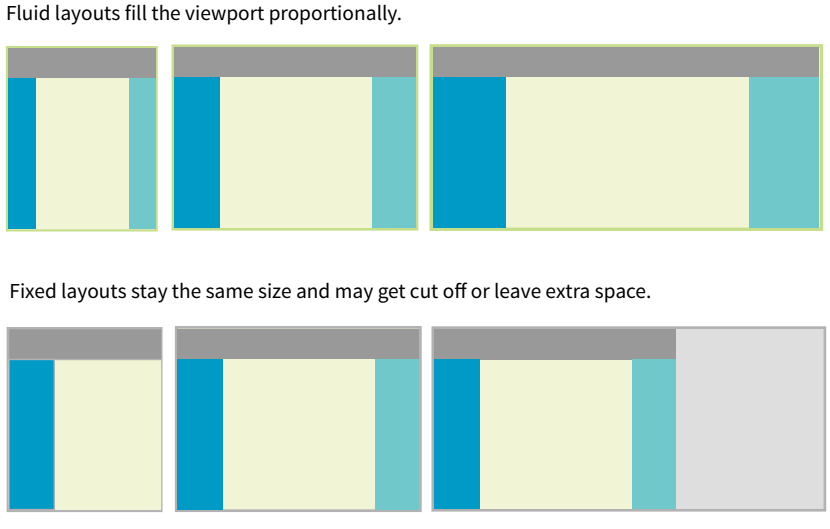

Fluid layouts fill the viewport proportionally.
Fixed layouts stay the same size and may get cut off or leave extra space.

FIGURE 19-3. Fluid and fixed layout examples

control over the layout that they might have had in print and ensured that users across all platforms and browsers got a similar, if not exactly identical, rendering of the page.

As devices of different sizes began to become more common, it didn't take long to realize that it would be impossible to create separate fixed-width designs tailored to each one. Clearly, fluidity has the advantage. It is based on the intrinsic nature of the normal flow, so we're working *with* the medium here rather than against it—no need to worry about horizontal scrollbars or awkward empty space in the browser. Now that the fluid, responsive approach has become the norm and has made its way into the tools, templates, and services we use to create sites, it has become difficult to find examples of fixed-width sites in the wild.

> **BACK IN MY DAY...**
>
> When I got started in web design in 1993, the most common PC monitor size was a measly 640 × 480 pixels, unless you were a fancy-pants designer type with an 800 × 600 screen. My earliest web page designs had a fixed width of an adorable 515 pixels.

FLEXIBLE IMAGES

The second RWD ingredient is flexible images. Every now and then, a solution is simple. Take, for example, the style rule required to make images scale down to fit the size of their container:

```
img {
  max-width: 100%;
}
```

That's it! When the layout gets smaller, the images in it scale down to fit the width of their respective containers. If the container is larger than the image—for example, in the tablet or desktop layouts—the image does not scale larger; it stops at 100% of its original size (FIGURE 19-4).

```
img { max-width: 100%; }
```

FIGURE 19-4. Setting the `max-width` of inline images allows them to shrink to fit the available space but not to grow larger than their actual size.

If you have provided **width** and **height** attribute values in the **img** element (they help the browser draw the layout of the page faster), you also need to set the **height** to **auto** in the style rule to maintain the image's aspect ratio:

```
img {
  max-width: 100%;
  height: auto;
}
```

Responsive Images

But wait—things are never *that* simple, right? While it may be tempting to just provide the largest image size you've got and let it shrink down for smaller devices, it's important to avoid serving unnecessarily large images (and all their data!) to small devices.

Thankfully, we have responsive image markup that allows us to do just that. I briefly introduced responsive images in **Chapter 7, Adding Images**, but to recap, the **picture** element and **srcset** and **sizes** attributes for the **img** element provide the ability to:

- Serve high-resolution images to high-density monitors only
- Provide images of varying dimensions aimed at different screen sizes
- Provide alternative versions of an image (e.g., cropped differently or with less detail) based on screen size
- Provide alternative image formats with smaller file sizes to browsers that can use them

This would be a good time to read the article "Responsive Image Markup," available at ***learningwebdesign.com/articles***, that I suggested saving for later. For now, just be aware that in the responsive design workflow, it is common to create multiple images at a range of sizes and then use markup to get them to the right devices (or use a service to do it for you!). I'll cover the image production component of responsive images in **Chapter 27, Image Production Techniques**. I also recommend the article "The Ultimate Guide to Responsive Images on the Web" by Anna Monus, which includes responsive techniques for embedded images and images used as CSS backgrounds (***debugbear.com/blog/responsive-images***).

Before we move on to the final RWD ingredient, why not give flexible images a try in EXERCISE 19-1?

EXERCISE 19-1. Making sure images fit

```
img {
  max-width: 100%;
}
```

FIGURE 19-5. Before and after adding the flexible image style rule

This exercise gives you a chance to see for yourself how setting the **max-width** on an image to 100% affects the image's behavior in the layout. Open the document *flexible-images.html* (available with the materials for this chapter) in your browser. There's not much to it—just three large images and some headings and paragraphs to give them context. The images are 1,200 pixels wide, so when the browser window is very wide, you should be able to see the whole thing. Try making the browser window narrower, and you will find that the images stay the same size and the right edges are no longer in view. You now know how to fix that!

Open *flexible-images.html* in a text or code editor and add the magic image flexible-izer to the style sheet:

```
img {
  max-width: 100%;
}
```

Save the document, reload it in the browser, and try resizing the window again. The widths of the images should change along with the size of the browser window (FIGURE 19-5). That's the "flexible image" ingredient in the original RWD recipe at work.

This is not, however, an example of the responsive image technique I mentioned in this section, because we are serving giant images to all users, including those with small devices. With responsive markup and a set of images at a range of sizes, you can fix that problem as well.

QUERY MAGIC

Now we get to the real meat of responsive design—queries! A query tests the browser or an element for certain conditions and then applies styles only when those conditions are met. They are what make it possible to send a one-column layout to small screens and a multicolumn layout to larger screens on the fly.

We now have two types of queries to help us make pages and components responsive. **Media queries** test for the type of media (screen or print) and various features of the device, like viewport width, resolution, other device capabilities, and even some user preferences related to motion and color. **Container queries** test for the width of a single containing element rather than the entire viewport. Container queries are still fairly new, and browser support is just gaining steam as of this writing, but they're quickly becoming the go-to tool for responsive design because in many cases the size of the element in the layout is more meaningful than the size of the viewport itself. We'll look at those shortly, but we're going to start with an overview of tried-and-true (and reliably supported) media queries.

Media Queries

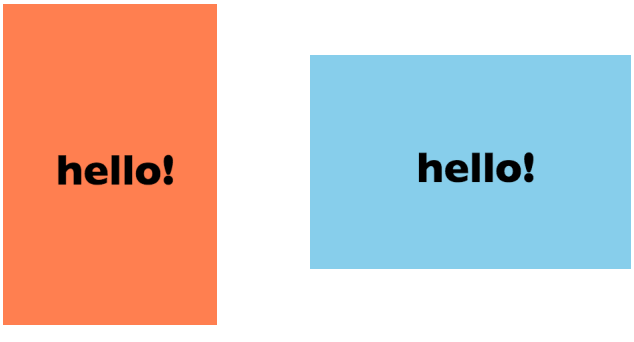
Media queries were our first tool for altering designs based on the viewing environment. The standard media query syntax includes a media type followed by a particular feature and a value for which to test. The test criteria are followed by a set of curly braces that contain the styles to apply if the test is passed. The structure of a media query as used within a style sheet looks like this:

```
@media type and (feature: value) {
    /* styles for browsers that meet these criteria */
}
```

A media query includes a media type followed by a particular feature and a value for which to test.

Let's clarify that with an example. The following media queries look at whether the viewport is on a screen and in **`landscape`** (horizontal) or **`portrait`** (vertical) orientation. When the query detects that the viewport is in landscape mode, the background color of the page is "skyblue"; when it's in portrait orientation, the background is "coral" orange (FIGURE 19-6). If this were displayed on a smartphone, the colors would change as the phone tilts from vertical to horizontal and back again. This isn't a very practical design choice, but it does provide a very simple illustration of media queries at work:

```
@media screen and (orientation: landscape) {
  body {
     background: skyblue;
     }
}

@media screen and (orientation: portrait) {
  body {
    background: coral;
    }
}
```



When the viewport is in portrait mode, the background color is "coral."

When the viewport is in landscape mode, the background color is "skyblue."

FIGURE 19-6. Changing the background color based on the orientation of the viewport with media queries

WARNING

Having style declaration curly braces nested inside media query curly braces can get a little confusing. Be sure that you have the right number of braces and nest them properly. Careful indenting is helpful. Many code editing programs also use color coding to help you keep them straight.

■ GOOD TO KNOW

Queries are always at work, even after the page has initially loaded. If the viewport changes—for example, if a user turns their phone from portrait to landscape orientation or resizes their desktop browser window—the query runs again and applies the styles appropriate for the new width.

Media types

Media types, as included in the first part of a query, were introduced in CSS2 as a way to target styles to particular media. For example, this **@media** rule delivers a set of styles only when the document is printed (it does not test for any specific features or values):

```
@media print {
  /* print-specific styles here */
}
```

This makes it easy to design the layout and typography and even show or hide particular content elements especially for print, and to keep the print-appropriate design from displaying on screens.

The current defined media types are **all**, **print**, and **screen** (see **Note**). If you are designing for screens, the media type is optional. Including it doesn't hurt (I'll be including the screen **media** type for the sake of clarity in the examples), but you can omit it if you prefer, as in this syntax example:

```
@media (feature: value) {
  }
```

NOTE

CSS2 also defined the media types **aural**, **handheld**, **braille**, **embossed**, **projection**, **speech**, **tty**, *and* **tv**, *but they have been deprecated in the Media Queries Level 4 and 5 specs and are discouraged from use in favor of targeting specific features.*

Media features

Media queries can also test for a particular *feature* of a viewport or device. In the previous example (FIGURE 19-6), we tested the orientation—landscape or portrait—of the device. For responsive design purposes, the most common feature to test for is the viewport width. You can also test for a minimum width (**min-width**) and maximum width (**max-width**).

Minimum-width queries are your go-to for creating mobile-first responsive design. They allow you to provide styles that kick in only when the viewport is the width you specify or wider. With this approach, you can set the baseline styles for narrow screens first, and then introduce new styles and make adjustments as needed for wider screens. Here's the syntax:

```
@media screen and (min-width: value) {
  /* styles for when the width is at least the provided value */
}
```

The following simple example displays headline fonts in a fancy cursive font only when the viewport is 30rem or wider (see **Note**)—that is, when there is enough space for the font to render legibly. Viewports that do not match the query (because they are narrower than 30rem) use a simple serif font. Notice that the query appears after the default style rule:

```
h2 {
  font-family: serif;
}

@media screen and (min-width: 30rem) {
  h2 {
    font-family: 'Lobster', cursive;
  }
}
```

NOTE

Using the relative unit **rem** *in a media query bases breakpoints on the user's default font size setting. A rem is 16 pixels by default, but a user can make it larger in the browser settings if they have limited vision. Basing breakpoints on font size has the advantage of keeping the page in proportion and line lengths in check. One potential downside is that if the user's font setting is very large, they may see the mobile version of the site even on a large desktop (although it's arguable whether that's a bad thing). There are sometimes good reasons for using* **px** *or* **em** *in media queries, so use what makes sense for your content.*

The complete list of device features you can detect with media queries appears in TABLE 19-1. Note that unless otherwise indicated, features labeled "Level 5 only" have partial or no browser support as of this writing. Be sure to check *CanIUse.com* for updated support statistics.

TABLE 19-1. Media features you can evaluate with media queries (Media Queries Level 5)

Feature	Description
Viewport/page characteristics	
`width`	The width of the display area (viewport). Also **`min-width`** and **`max-width`**.
`height`	The height of the display area (viewport). Also **`min-height`** and **`max-height`**.
`aspect-ratio`	The viewport's width divided by its height. Example: **`aspect-ratio: 16 / 9`**.
`orientation`	Whether the device is in **`portrait`** or **`landscape`** orientation.
`overflow-block`	How the device handles content that overflows the viewport along the block axis.
`overflow-inline`	Whether the content that overflows the viewport along the inline axis can be scrolled.
`horizontal-viewport-segments`	*[Level 5 only]* Whether the viewport is segmented in the horizontal direction (such as a fold or hinge between separate displays).
`vertical-viewport-segments`	*[Level 5 only]* Whether the viewport is segmented in the vertical direction (such as a fold or hinge between separate displays).
`display-mode`	*[Level 5 only]* How the page or application is presented within the context of an operating system. Values include **`fullscreen`**, **`standalone`**, **`minimal-ui`**, and **`browser`**.
Display quality features	
`resolution`	The density of pixels in the device. This is increasingly relevant for detecting high-resolution displays.
`scan`	Whether a **`tv`** media type uses progressive or interlace scanning. (Does not accept **`min-`**/**`max-`** prefixes.)
`grid`	Whether the device uses a grid-based display, such as a terminal window. (Does not accept **`min-`**/**`max-`** prefixes.)
`update`	How quickly (if at all) the output device modifies the appearance of the content.
`environment-blending`	*[Level 5 only]* Characteristics related to the opacity of the user's display, such as for interfaces overlaid on the natural environment, like a head-up display in a car. Values are **`opaque`**, **`additive`**, and subtractive.
Color features	
`color`	The bit depth of the display. For example, **`color: 8`** tests whether the device has at least 8-bit color.
`color-index`	The number of colors in the color lookup table.
`color-gamut`	The approximate range of colors that are supported by the user agent and output device.
`monochrome`	The number of bits per pixel in a monochrome device.
`dynamic-range`	*[Level 5 only]* A combination of the maximum brightness, color depth, and contrast of the display. Values are **`high`** and **`standard`**.
`inverted-colors`	*[Level 5 only]* Whether the content is displayed normally (none) or with colors inverted (inverted).

TABLE 19-1. Continued

Feature	Description
Interaction features	
`pointer`	Whether the primary input mechanism is a pointing device, and how accurate it is. Values are **`none`**, **`course`**, and **`fine`**.
`hover`	Whether the input mechanism allows the user to hover over elements. Values are **`none`** and **`hover`**.
`any-pointer`	Whether any available input mechanism is a pointing device, and how accurate it is. Values are **`none`**, **`course`**, and **`fine`**.
`any-hover`	Whether any available input mechanism allows hovering. Values are **`none`** and **`hover`**.
`nav-controls`	*[Level 5 only]* Whether the user agent (browser) has navigation controls such as a back button. Values are **`none`** and **`back`**.
User preference features Browsers have supported all of these features since 2020, with the exception of **`prefers-reduced-data`**.	
`prefers-reduced-motion`	*[Level 5 only]* Whether the user has requested that the system minimize animation and motion on the page. Values are **`no-preference`** and **`reduce`**.
`prefers-reduced-transparency`	*[Level 5 only]* Whether the user has requested that the system minimize transparent and translucent layer effects. Values are **`no-preference`** and **`reduce`**.
`prefers-contrast`	*[Level 5 only]* Whether the user has requested more or less contrast on the page. Values are **`no-preference`**, **`less`**, **`more`**, and **`custom`**. (Browser support since 2022.)
`forced-colors`	*[Level 5 only]* Whether the user has chosen to use a limited color palette on the page. Values are **`none`** and **`active`**.
`prefers-color-scheme`	*[Level 5 only]* Whether the user has chosen a dark or light color theme. Values are **`light`** and **`dark`**.
`prefers-reduced-data`	*[Level 5 only]* Whether the user has a preference for being served a lower-data version of the content. Values are **`no-preference`** and **`reduce`**.

Media Queries for Accessibility

As you will see in **Chapter 20, Transitions, Transforms, and Animation**, we have many options for adding motion to interfaces. Unfortunately, transitions, motion, flashing lights, and fancy scrolling effects can cause serious problems for people with vestibular disorders and visual processing issues. For more insight, read Eileen Webb's article "Your Interactive Makes Me Sick" (*source.opennews.org/articles/motion-sick/*), which describes her experience with web special effects.

In an effort to improve accessibility and provide a better experience for these users, Media Queries Level 5 added the **`prefers-reduced-motion`** feature query. This query detects whether the user has set their operating system preferences to "reduce motion," so you can provide alternative styles that reduce or omit those visual effects.

There are also feature queries for accommodating users' preferences for increased contrast, a limited color palette, limited transparency, and whether to use a dark or light theme. To get up to speed for with user preference feature queries, start with these articles:

- "Designing with Reduced Motion for Motion Sensitivities" by Val Head (*oreil.ly/xz-Q-*)
- "CSS and Accessibility: Inclusion Through User Choice" by Carie Fisher (*oreil.ly/nDR56*)
- "Complete Guide to User Preference Media Features in CSS" by Michael Gearon (*mgearon.com/css/user-preference-media-features/*)

How to use media queries

You can use media queries within a style sheet or to conditionally load external style sheets. Media queries may not be used with inline styles.

Within a style sheet

The most common way to utilize media queries is to use an `@media` ("at-media") rule right in the style sheet. The examples in this chapter so far have all used this technique.

When you use media queries within a style sheet, the order of rules is very important. Because rules later in the style sheet override the rules that come before them, you need to be careful about the order of queries containing the same property/value declarations.

One solid strategy is to specify the baseline styles that serve as a default, and then override specific rules as needed to optimize for alternate viewing environments with queries *after* the baseline rules in the stylesheet. The best practice is to set up styles for small screens and then introduce styles for increasingly larger screens later in the style sheet using `min-width`.

That's exactly what I did in the headline font–switching example earlier. The first `h2` rule sets a baseline experience with a local serif font, which then gets enhanced for larger screens with a media query.

With external style sheets

For large or complicated sites, developers may choose to put styles for different devices into separate style sheets and call in the whole *.css* file when certain conditions are met. One method is to use the `media` attribute in the `link` element to conditionally load separate *.css* files. In this example, the basic styles for a site are requested first, followed by a style sheet that will be used only if the device's screen is more than 1,000 pixels wide (and if the browser supports media queries):

```
<head>
  <link rel="stylesheet" href="styles.css">
  <link rel="stylesheet" href="2column-styles.css"
        media="screen and (min-width:1000px)">
</head>
```

Some developers find this method helpful for managing modular style sheets, but it comes with the disadvantage of requiring extra HTTP requests for each additional *.css* file. It is also less maintainable, requiring changes to multiple documents if anything changes. Be sure to provide only as many links as necessary (perhaps one for each major breakpoint), and rely on `@media` rules within style sheets to make minor adjustments for sizes in between.

Similarly, you can carry out media queries with **@import** rules that pull in external style sheets from within a style sheet. Notice that the word "media" does not appear in this syntax, only the type and query:

```
<style>
  @import url("default-styles.css">;
  @import url("/wide-styles.css") screen and (min-width: 1000px);
  /* other styles */
</style>
```

Now would be a good time for you to give media queries a try to improve the Black Goose Bakery page in EXERCISE 19-2.

Line Length Trick

A good trigger for deciding when to adjust the layout is to look at text line lengths. Lines of text that are too stubby or too long are difficult to read, so you should aim for optimal line lengths of 45 to 75 characters, including spaces. If your text lines are significantly longer, it's time to make changes to the layout, such as increasing the margins or introducing an additional column. You might also increase the font size of the text to keep the character count in the desired range.

Clarissa Peterson introduces a neat trick for testing line lengths in her book *Learning Responsive Web Design* (O'Reilly): put a **span** around the 45th to 75th characters in the text and give it a background color, as I've done in this exercise. That way, you can easily check whether the line breaks are happening in the safe zone at a glance. Of course, you'll want to remove this line length hint before making the site public.

We'll use the line length trick in EXERCISE 9-2, and it's visible in FIGURE 19-8.

How Wide Is the Viewport?

It is common for designers and developers to gradually increase the width of their browser window until they see that a breakpoint is needed. But how can you tell how wide the window is in pixels?

Fortunately, Chrome, Safari, and Firefox have tools that show you how a page looks at specific viewport dimensions. In the responsive mode (or view), you get a resizable window-in-a-window that can be set to standard device sizes or resized manually. The pixel dimensions are displayed as the viewport resizes.

To access Chrome's device toolbar, shown in FIGURE 19-7, select **View → Developer → Developer Tools**, and then look for the "Toggle device toolbar" icon on the left side of the menu bar. Safari's Responsive Design Mode is accessible via **Develop → Enter Responsive Design Mode**. In Firefox, access Responsive Design Mode via the Web Developer Tools (**Tools → Web Developer → Responsive Design Mode**). They all work similarly, but you may find you prefer one browser's user interface over another's.

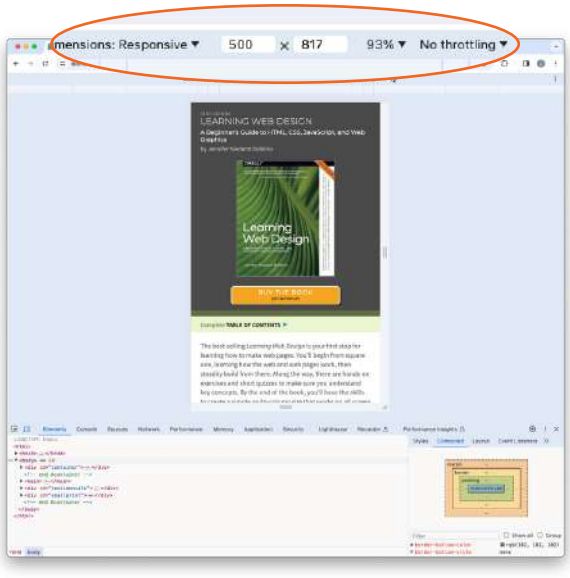


FIGURE 19-7. Chrome's responsive viewport mode in Developer Tools displays the pixel width of the display area.

EXERCISE 19-2. Making the bakery home page responsive

We've done a lot of work on the Black Goose Bakery site over the last few chapters, but the resulting site works best on large screens. In this exercise, we're going to back up a few steps and build it again using a small-screen-first strategy, making changes to layout, navigation, typography, and more at strategic breakpoints.

I've done the heavy lifting of writing the necessary styles for each breakpoint, but I'll talk you through each step and share the reasoning for the changes. The starting HTML file (*bakery-rwd.html*) and style sheet (*bakery-rwd.css*) as well as the finished style sheet (*bakery-rwd-finished.css*) are provided with the materials for this chapter.

Getting started

Open *bakery-rwd.html* in a browser with a responsive view (see the sidebar **"How Wide Is the Viewport?"**) so you can expand the viewport window and watch the changing pixel dimensions. FIGURE 19-8 shows the page at 320 pixels wide with the default, narrow-screen styles that will be the starting point for this design.

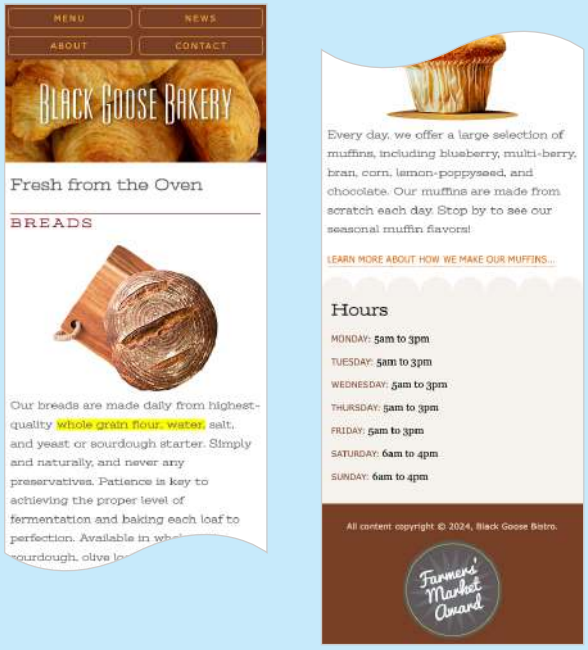

MENU
NEWS
ABOUT
CONTACT
BLACK GOOSE BAKERY
Fresh from the Oven
BREADS
Our breads are made daily from highest-quality whole grain flour, water, salt, and yeast or sourdough starter. Simply and naturally, and never any preservatives. Patience is key to achieving the proper level of fermentation and baking each loaf to perfection. Available in wh
sourdough, olive lo
Every day, we offer a large selection of muffins, including blueberry, multi-berry, bran, corn, lemon-poppyseed, and chocolate. Our muffins are made from scratch each day. Stop by to see our seasonal muffin flavors!
LEARN MORE ABOUT HOW WE MAKE OUR MUFFINS...
Hours
MONDAY: 5am to 3pm
TUESDAY: 5am to 3pm
WEDNESDAY: 5am to 3pm
THURSDAY: 5am to 3pm
FRIDAY: 5am to 3pm
SATURDAY: 6am to 4pm
SUNDAY: 6am to 4pm
All content copyright © 2024, Black Goose Bistro.
Farmers' Market Award

FIGURE 19-8. The small-screen design is our starting point.

The content of the page is the same as in previous chapters, but if you worked on the exercises in **Chapter 18, Grid Layout**, you'll notice that I've changed a few styles to make the initial layout suitable for small screens. Here is an overview of this baseline design:

- **Layout**: The page has a one-column layout for small screens. There are no borders around the main text area, and the Hours section has a scalloped edge on the top instead of the side. That maintains the look and feel, but it's more appropriate when the sections are stacked.
- **Navigation:** The navigation menu, which was created with Flexbox, couldn't flex small enough to fit across a small screen. To make it fit, I turned on wrapping (`flex-wrap: wrap`) and set the width of each `li` to 50% so there would be two on each row, able to both grow and shrink as needed (`flex: 1 1 50%`).
- **Conditional header text:** The tagline (`header p`) was taking up a lot of vertical space, and I decided it wasn't critical, so I hid the paragraph (`display: none`); I'll make it visible again when there is more room.
- **Images:** I set the `img` elements for the bread and muffin images to `display: block` so they have the full width of the viewport to themselves with no text sneaking in next to them. Setting the side margins to `auto` keeps them centered horizontally.
- **Miscellaneous:**
  - The award appears at the bottom of the page because there is not enough space for it to be positioned at the top.
  - In the first paragraph of the HTML file, a `span` from the 45th to 75th characters has been given a yellow background color to reveal when the line lengths get too long (see the sidebar **"Line Length Trick"**).

Fixing the navigation

Now we can start tailoring the design for other screen sizes. Using a responsive view tool, you can resize the viewport and get an instant readout of the dimensions of the window. Give it a try in your browser. Keep making the viewport wider, and you'll see that some things look OK, but some things start looking awkward pretty quickly.

One thing that looks awkward right away is the stacked navigation at the top. I'd like it to switch to one centered line as soon as there is room, which happens when the viewport is approximately 450 pixels wide (FIGURE 19-9).

EXERCISE 19-2. Continued

Are you ready to write your first media query? Open the style sheet (*bakery-rwd.css*) in a text editor. Remember that media queries need to come after other rules for the same declaration, so to keep this exercise simple, we'll add them at the end of the style sheet, before `</style>`. Add this query as you see it here. Remember to make sure you have the right number of opening and closing curly braces and that they are nested properly:

```
@media screen and (min-width: 450px) {
  nav ul li {
    flex: none;
  }
  nav ul {
    justify-content: center;
  }
}
```

This tells the browser that when the page is on a screen and the viewport is 450 pixels or wider, it should set the `flex` property of the navigation menu's list items to `none`. The `none` keyword is equivalent to `flex: 0 0 auto`, so the items are not allowed to grow or shrink and will be sized based on their content. I've centered the flex container by setting `justify-content: center`.

Save the style sheet and reload the page in your browser. Try resizing the viewport to see how it looks at wider sizes. This centered arrangement should work for even the widest of screens, so our navigation menu is all set. If we'd had additional elements in our navigation section, such as a logo and a search box, we might have found it best to create a few different arrangements over a number of breakpoints.

Before breakpoint change

After

FIGURE 19-9. The navigation started to look awkward, so I added a breakpoint at 450 pixels to switch it to one line.

Floating images

As we continue to make the viewport gradually wider, the main images start looking very lonely all by themselves on their lines, and there is room to start wrapping text around them again at about 500 pixels wide. Let's take care of that awkward whitespace by floating the images to the left once the screen reaches 500 pixels (FIGURE 19-10):

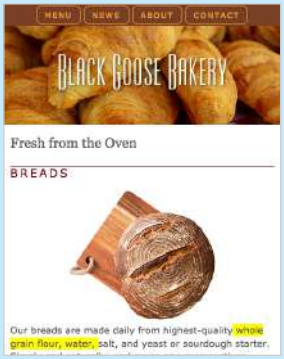
```
@media screen and (min-width: 500px) {
  main img {
    float: left;
    margin: 0 16px 16px 0;
  }
}
```

NOTE

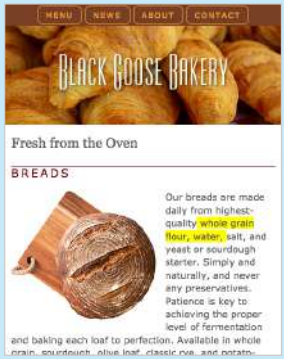
If you like, you can include the CSS shapes from **Chapter 16, Floating and Positioning***, for a more interesting text wrap. I've omitted them here for brevity.*

Text and typography

Once the viewport gets to about 600 pixels wide, there is enough room to introduce some embellishments, such as the tagline in the header, so we'll set that to display again and specify its margins on all four sides. We'll take advantage of the extra space to add the rounded border and 20px margin around the main text area to bring it closer to the original brand identity for the site. The result is an enhanced one-column layout that is well suited for tablet-sized devices (FIGURE 19-11).



A breakpoint is needed to fill in the awkward space around the image.


MENU
NEWS
ABOUT
CONTACT
BLACK GOOSE BAKERY
Fresh from the Oven
BREADS
Our breads are made daily from highest-quality whole grain flour, water, salt, and yeast or sourdough starter. Simply and naturally, and never any preservatives. Patience is key to achieving the proper level of fermentation and baking each loaf to perfection. Available in whole

At 500 pixels wide, the image is floated to the left.

FIGURE 19-10. The images float left once there is enough width to accommodate wrapping text.

EXERCISE 19-2. Continued

Here is the media query for the 600-pixel breakpoint. Add this to the bottom of the style sheet, after the other two queries:

```
@media screen and (min-width: 600px) {
  header p {
    display: block;
    margin: -12px 16px 0 16px;
    }
   main {
    border: double 4px #EADDC4;
    border-radius: 25px;
    margin: 20px;
  }
}
```

Multicolumn layout

As you continue to make the viewport wider, if you pay attention to the yellow highlighted span of characters, you will see that the text line is growing longer than 75 characters. We could increase the font size or the margins, but this is a good point to introduce a second column to the layout. If you aren't targeting a specific device, the exact positioning of the breakpoint is subjective. I've chosen 940 pixels as the point above which the page gets a columned layout.

I've simply taken the grid styles from the previous chapter and reapplied them here. On the **aside** element, I moved the scalloped background graphic to the left edge. In addition, I set a maximum width of 1200px on the container and set its side margins to **auto**, so if the browser window is wider than 1,200 pixels, the layout will stay a fixed width and get centered in the viewport. Finally, the award graphic is absolutely positioned at the top of the page now that there's enough room (FIGURE 19-12).

FIGURE 19-11. This medium-size layout is well suited for tablet-sized devices.

NOTE

The highlighted background on the "linelength" **span** *should be turned off before you publish, but I've left it visible in the figures so you can see how the line length is faring across layouts.*

Add this final media query at the end of the style sheet. It's a lot to type, so you can copy and paste it from the final exercise in **Chapter 18** (that's what I did) and make a few tweaks to the **#container** and **#aside** rules, as shown here:

```
@media screen and (min-width: 940px) {
  #container {
    display: grid;
    grid-template-rows: auto min-height 5em;
    grid-template-columns: minmax(25em, 1fr) 16em;
    grid-template-areas:
      "banner banner"
      "main hours"
      "footer footer";
    max-width: 1200px;
    margin: 0 auto;
    position: relative;
  }
```

EXERCISE 19–2. Continued

```
    header {
      grid-area: banner;
    }
    main {
      grid-area: main;
    }
    aside {
      grid-area: hours;
      background: url(images/scallop.png) repeat-y left top;
      background-color: #F6F3ED;
      padding: 16px;
      padding-left: 45px;
    }
    footer {
      grid-area: footer;
    }
    #award {
      position: absolute;
      top: 30px;
      left: 50px;
    }
  }
```

And we're done! Is this the most sophisticated responsive site ever? Nope. Is there even more we could do to improve the design at various screen sizes? Certainly! But now you should have a feel for what it's like to start with a small-screen design and make changes to individual components so they are optimized for increasingly larger sizes. Consider it a modest first step toward future adventures in responsive design.

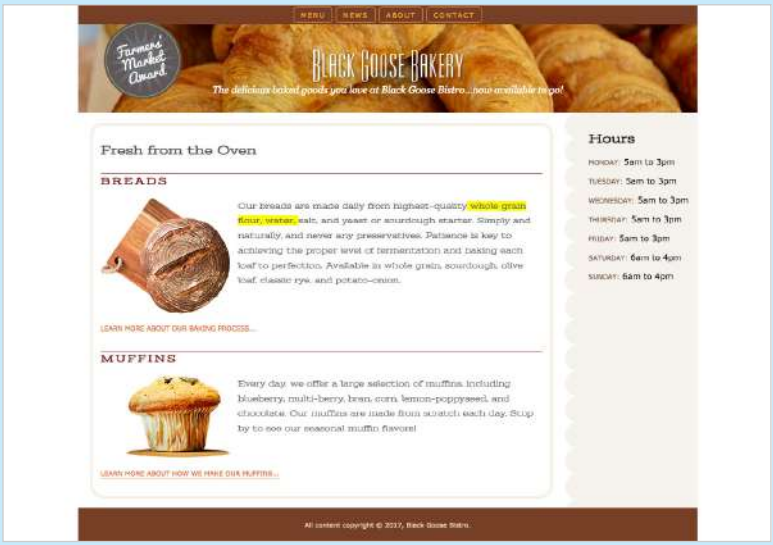

MENU
NEWS
ABOUT
CONTACT
Farmers Market Award
Black Goose Bakery
The delicious baked goods you love at Black Goose Bistro...now available to go!
Fresh from the Oven
BREADS
Our breads are made daily from highest-quality whole grain flour, water, salt, and yeast or sourdough starter. Simply and naturally, and never any preservatives. Patience is key to achieving the proper level of fermentation and baking each loaf to perfection. Available in whole grain, sourdough, olive loaf, classic rye, and potato-onion.
LEARN MORE ABOUT OUR BAKING PROCESS...
MUFFINS
Every day, we offer a large selection of muffins including blueberry, multi-berry, bran, corn, lemon-poppyseed, and chocolate. Our muffins are made from scratch each day. Stop by to see our seasonal muffin flavors!
LEARN MORE ABOUT HOW WE MAKE OUR MUFFINS...
Hours
MONDAY: 5am to 3pm
TUESDAY: 5am to 3pm
WEDNESDAY: 5am to 3pm
THURSDAY: 5am to 3pm
FRIDAY: 5am to 3pm
SATURDAY: 6am to 4pm
SUNDAY: 6am to 4pm
All content copyright © 2017, Black Goose Bistro.

FIGURE 19-12. The two-column grid layout is appropriate for viewports over 940 pixels wide. On very wide screens, as shown here, the container stops expanding at 1,200 pixels wide and is centered horizontally.

Container Queries

Now it's time to look at the other type of query we have at our disposal. As the name suggests, **container queries** apply styles based on the size of an element's *containing element*, not the entire viewport. Container queries make page components more self-contained and able to work in a variety of contexts.

In many cases, the width the element occupies in the layout is more relevant than the size of the entire window, so having the ability to make design decisions based on component size is a real advantage. To see what I mean, take a look at the Flexbox example in FIGURE 19-13. The third flex item is narrow on narrow screens, of course. In slightly wider viewports, its flex setting allows it to expand much wider; however, on even wider viewports, it gets narrow again because it has room to line up with the others.

The third item is narrow on narrow viewports.

It grows wider on a wider viewport.

It is narrow again on the widest viewports.

FIGURE 19-13. The overall viewport width is not always the best predictor of how elements should be styled in responsive layouts.

There are two steps to setting up a container query:

1. Identify an element as the container to watch with the `container-type` property.
2. Apply conditional styles to its descendent elements with an `@container` rule.

Identifying the container

Take a look at the middle screenshot in FIGURE 19-13. When the flex item is wide, there is certainly enough space for the text and image to appear side by side, which would eliminate some awkward whitespace (FIGURE 19-14). Let's look at the code to make that happen.

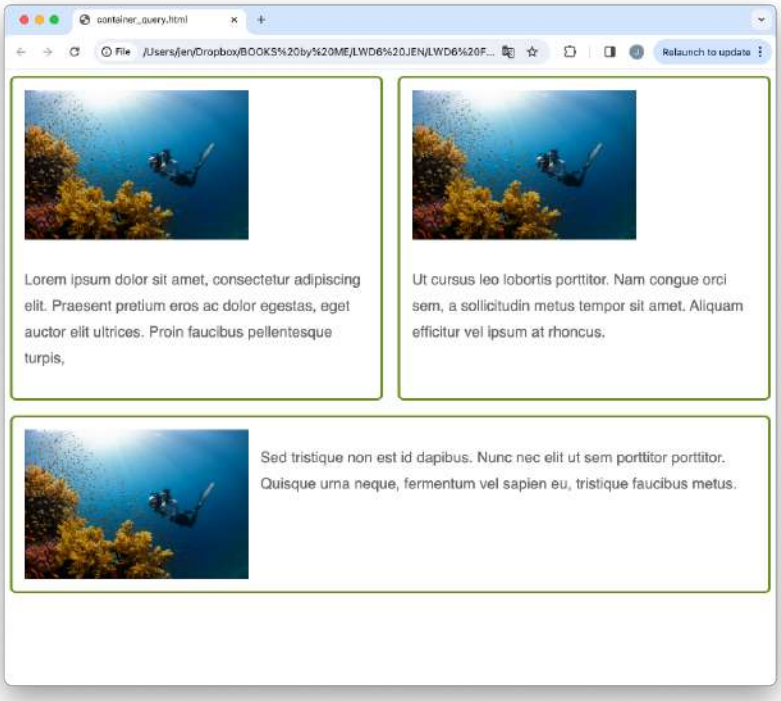

container_query.html
Relaunch to update
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent pretium eros ac dolor egestas, eget auctor elit ultrices. Proin faucibus pellentesque turpis,
Ut cursus leo lobortis porttitor. Nam congue orci sem, a sollicitudin metus tempor sit amet. Aliquam efficitur vel ipsum at rhoncus.
Sed tristique non est id dapibus. Nunc nec elit ut sem porttitor porttitor. Quisque urna neque, fermentum vel sapien eu, tristique faucibus metus.

FIGURE 19-14. Using a container query, we can move the text beside the image when the flex item is wider than 600px.

We're starting with a **div** (**#flexbox**) that is a flex container for three "listing" **div**s. Each listing contains a content **div** that is also a flex container with its direction set to **column**, so the image and descriptive text stack up nicely. Our goal is to allow them to appear in a row when the listing is wide enough to do so comfortably. Here is an abbreviated look at the markup:

```
<div id="flexbox">
  <div class="listing">
    <div class="content">
      <p class="photo"><img src="photo.jpg" width="300" →
height="200"></p>
      <p class="description">...</p>
    </div>
  </div>
  <!-- two more .listing divs with the same structure -->
</div> <!-- end of #flexbox div -->
```

We want the browser to pay attention to the width of the **.listing** elements, so we apply the **container-type** property to those **div**s to make them official "containers" (this is called creating a **containing context**):

```
.listing {
  container-type: inline-size;
}
```

There are three values for the **container-type** property, but the one that is relevant for adapting to widths that change on the fly is **inline-size** (see **Note**). The other values are **size** (which requires **width** and **height** values on the container and works somewhat differently) and **normal** (which doesn't create a containing context, but may one day work for contextual styling).

You can also give the container a name with the **container-name** property. Being able to call out the container by name eliminates confusion when containing contexts are nested. If no name is provided, the container query uses the closest container in the document hierarchy. The value is case-sensitive, so "resource" is not interchangeable with "Resource":

```
.listing {
  container-type: inline-size;
  container-name: resource;
}
```

Because the W3C is simply wild about shorthand properties, you can combine the type and name values with **container**. The **container-name** value goes first, followed by a forward slash and then the **container-type**. Both values must be present in order to use the **container** shorthand:

```
.listing {
  container: resource / inline-size;
}
```

Applying conditional styles with @container

OK, we've told the browser to keep an eye on the inline size (width) of the **.listing** elements; now we'll use a container query to base styles on that size (see **Note**). In this example, we are changing the **flow-direction** of the **.content div**s from **column** to **row** when their **.listing** containers are 600 pixels or wider:

```
@container (min-width: 600px) {
  .content {
     flex-direction: row;
  }
}
```

The **@container** rule is structured much like the media query we looked at previously. The test conditions go inside parentheses, and the styles that get applied if the conditions are met go between curly braces. Again, because the style rules have their own curly braces, take care to make sure everything is nested and closed correctly.

To call out a container by name in a query, include the name before the test criteria:

```
@container resource (min-width: 600px) {
  ...
}
```

NOTE

Just a little note about the term "inline" in the **inline-size** *value. As you learned in the Flexbox and Grid Layout chapters, newer CSS specs are moving toward logical terms based on writing direction. "Inline" corresponds to "width" for languages that are written horizontally. You'll see this come up in various contexts in newer modules;* **inline-size** *is just one example.*

NOTE

If you use a container query without first setting up a containing context with **container-type**, *the query will use the width of the viewport.*

The spec outlines a few other ways to write out the criteria, including using conditionals (`>`, `>=`, `<`, and `<=`), so we could also write the previous query like this:

```
@container resource (width > 600px) { ... }
```

There's definitely more to the container query story, but I'm afraid I'll need to leave that to the spec and your own research. Before I wrap up, though, there's one more exciting new feature to share.

Container query units

Chapter 11, Introducing Cascading Style Sheets, introduced the viewport units (**`vw`** and **`vh`**) that allow you to specify lengths in percentages of the viewport width and height. Well, now we can specify sizes relative to the size of the containing context as well by using the new **container query length units** introduced in CSS Containment Module Level 3.

Container-based units are very helpful for fluid typography because they ensure that font size is always relative to an element's container, not the whole viewport. (We've already seen how the viewport might not be the best basis for styling individual page components.) In this example, the font size of an **`h1`** is set to be 15% of the width of its established containing element by using the **`cqi`** (container query inline size) unit. Each unit equals 1% of the container's inline size (i.e., width):

```
h1 {
  font-size: 15cqi;
}
```

TABLE 19-2 lists the new container query units.

TABLE 19-2. Container query length units

Unit	Description
`cqw`	1% of a query container's width
`cqh`	1% of a query container's height
`cqi`	1% of a query container's inline size
`cqb`	1% of a query container's block size
`cqmin`	The smaller value of either **`cqi`** or **`cqb`**
`cqmax`	The larger value of either **`cqi`** or **`cqb`**

Browser support and fallbacks

Container queries and units are still fairly new as of this writing, only achieving full browser support in 2023. Be sure to check ***CanIUse.com*** for updated support information. You may also want to provide alternative styles to non-supporting browsers using feature detection (see the "CSS Feature Detection" section in **Chapter 21, More CSS Techniques**, for an explanation).

For further reading

I've really only scratched the surface of container queries here. Because they are emerging as a powerful tool for responsive design, you may want to do more exploration on your own, starting with these listed resources. Of course, practice is a good idea, so give them a try in EXERCISE 19-3.

- CSS Containment Module Level 3 (*w3.org/TR/css-contain-3*)
- "Getting Started with CSS Container Queries" by Michelle Barker, on the MDN Web Docs blog (*oreil.ly/vGvgb*)
- "How to Use Container Queries Now" (i.e., in 2024, when this helpful article on the Chrome Developer blog was written) by Philip Walton (*oreil.ly/xgdUB*)
- "A Primer on CSS Container Queries" by Stephanie Eckles (*oreil.ly/KPZGP*)

Media and container queries are our last code-centric topics in this chapter, but there is more to responsive design than just code. Next, I'll cover some general guidelines and best practices for making your sites responsive.

EXERCISE 19-3. Container queries

This exercise gives you a bit of hands-on experience making simple design tweaks based on container width. The starting document, *breads.html*, contains a flexbox with three "card" items featuring a selection of international breads. Each item contains a "content-wrapper" **`div`** that is also a flexbox, with an image and description. We'll be using container queries to make the images and descriptions stack when the card items are narrow, and we'll increase the font size of the headings when the card items are wide.

1. Open *breads.html* in an up-to-date browser and resize the window to see how the cards behave. You may want to view the page in your browser's responsive mode so you can see the pixel measurements as you resize the panel. As in the example earlier in this section, the items' widths are not always proportional to the viewport width, so this is a good opportunity to use container queries.
2. We'll make style changes based on the width of the item cards (**`div`**s with the class "item"), so find the style rule for **`.item`** and use the shorthand **`container`** property to set the containing context and give it the name "card":

```
.item {
  ...
  container: card / inline-size;
}
```

3. First, we'll make the images and descriptions stack vertically when the items are narrow. I played around with it and decided the change should kick in when the item is 400px wide or less, but you can try other values. Create a new **`@container`** query that changes the direction of the **`.content-wrapper`** flexboxes to **`column`** when the "card" container is less than 400px wide. We could test for the **`width`** in the query, but let's use the equivalent **`inline-size`**, to follow the CSS spec's movement toward logical, writing direction–based terms:

```
@container card (inline-size < 400px) {
  .content-wrapper {
    flex-direction: column;
  }
```

4. Next, we'll increase the size of the **`h2`** headings when the card items are more than 540px wide. Add the following container query to the style sheet (feel free to try your own width and font size values). When you are finished, refresh the page in your browser and resize the window to see the effects of the new styles.

```
@container card (inline-size > 540px) {
  h2 {
    font-size: 1.5rem;
  }
}
```

EXERCISE 19-3. Continued

WARNING

This has been a simple container query exercise, but know that they may cause unexpected behaviors in real scenarios. One quirk is that when you turn a flex item into a containing context, it no longer bases its width on its child elements. You must provide a width explicitly (in this exercise, the **flex-basis** *value did the trick). I recommend Kevin Powell's very helpful video tutorial "Container Queries Are Going to Change How We Make Layouts" (oreil.ly/gBJFl), in which he discusses this particular issue.*

5. As long as we're playing with font size in a container, why not try out container units? A **cqi** unit equals 1% of the container's **inline-size** (width). Change the **font-size** value to 3cqi, save the file, and reload the page in your browser. Resize the browser window larger and smaller, and pay attention to the way the **h2** headings size up gradually as the container gets wider. This is an example of fluid typography, which we will be discussing in more detail later in the chapter.

```
@container card (inline-size > 540px) {
  h2 {                                  }
    font-size: 3cqi;                   }
```

At this point, your page should look something like the one in FIGURE 19-15. If not, keep trying. Container queries can be tricky, as explained in the **Warning**.

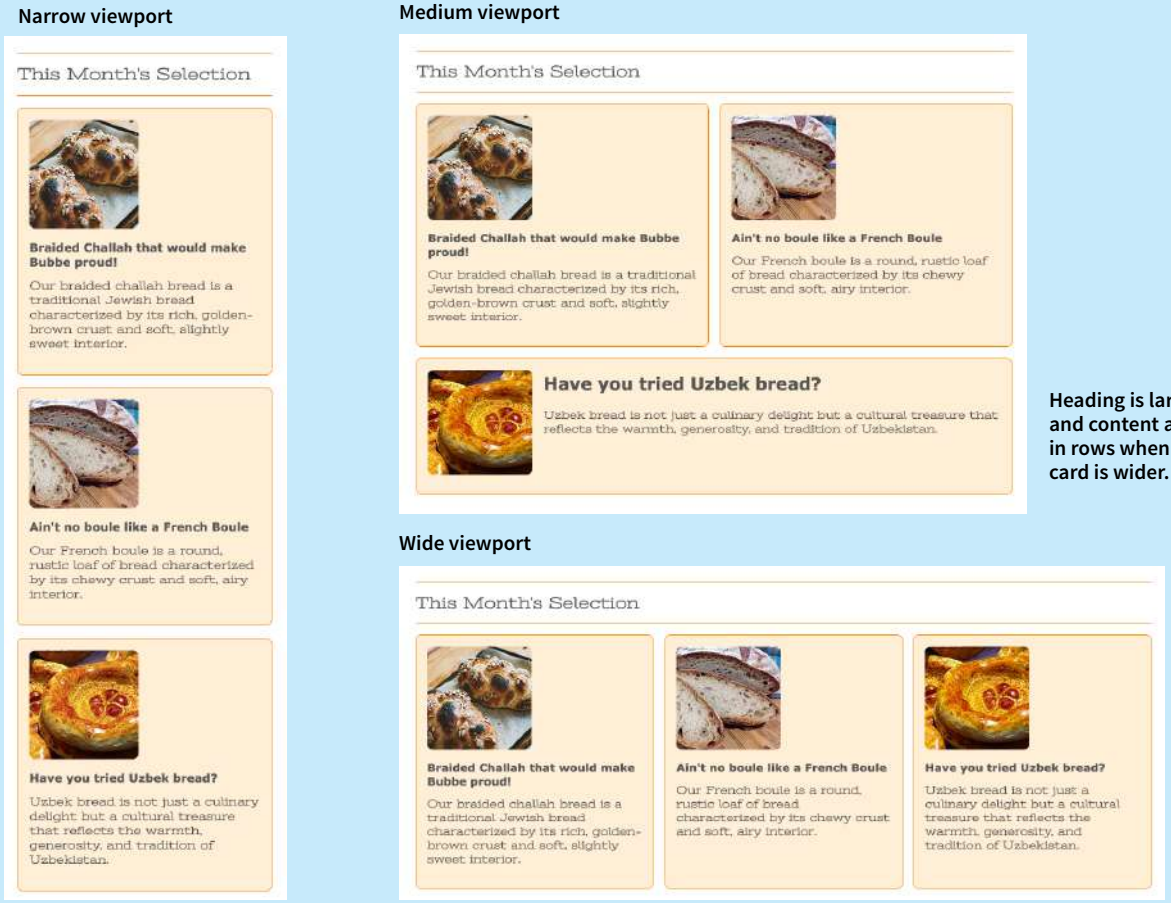

Narrow viewport
This Month's Selection
Braided Challah that would make Bubbe proud!
Our braided challah bread is a traditional Jewish bread characterized by its rich, golden-brown crust and soft, slightly sweet interior.
Ain't no boule like a French Boule
Our French boule is a round, rustic loaf of bread characterized by its chewy crust and soft, airy interior.
Have you tried Uzbek bread?
Uzbek bread is not just a culinary delight but a cultural treasure that reflects the warmth, generosity, and tradition of Uzbekistan.
Medium viewport
This Month's Selection
Braided Challah that would make Bubbe proud!
Our braided challah bread is a traditional Jewish bread characterized by its rich, golden-brown crust and soft, slightly sweet interior.
Ain't no boule like a French Boule
Our French boule is a round, rustic loaf of bread characterized by its chewy crust and soft, airy interior.
Have you tried Uzbek bread?
Uzbek bread is not just a culinary delight but a cultural treasure that reflects the warmth, generosity, and tradition of Uzbekistan.
Heading is larger and content aligns in rows when the card is wider.
Wide viewport
This Month's Selection
Braided Challah that would make Bubbe proud!
Our braided challah bread is a traditional Jewish bread characterized by its rich, golden-brown crust and soft, slightly sweet interior.
Ain't no boule like a French Boule
Our French boule is a round, rustic loaf of bread characterized by its chewy crust and soft, airy interior.
Have you tried Uzbek bread?
Uzbek bread is not just a culinary delight but a cultural treasure that reflects the warmth, generosity, and tradition of Uzbekistan.

FIGURE 19-15. The resulting page after making alignment and font size adjustments with container queries

DESIGNING RESPONSIVELY

We've covered the RWD nuts and bolts—now let's talk about some of the decisions designers make when creating responsive sites. I'll provide overviews, but you'll find much deeper explorations on these themes in the books and articles listed along the way. For now, I just want to raise your awareness of responsive strategies.

We've seen a few examples of content looking wonky when the browser gets very narrow or very wide. A three-column layout just doesn't fit, and text in an image may become unreadable when the page is scaled down to fit a 400-pixel-wide screen. At the other end of the spectrum, the line length in single-column layouts becomes too long to read comfortably when the viewport fills a desktop monitor. For many aspects of a web page, one size does not fit all. As designers, we need to pay attention to where things fall apart and set breakpoints to "preserve the integrity" of the elements (as Ethan Marcotte so nicely puts it).

In the broadest of strokes, these are the tricky bits to keep optimized over a wide range of viewport sizes:

- Breakpoints
- Content hierarchy
- Layout
- Typography
- Navigation
- Images
- Special content such as tables, forms, and interactive features

Choosing Breakpoints

A **breakpoint** is the point at which a query introduces a style change. When you specify `min-width: 800px` in a query, you are saying that 800 pixels is the breakpoint at which those particular styles should kick in. FIGURE 19-16 shows some of the breakpoints at which Etsy's site circa 2018 made both major layout changes and subtle design tweaks on the home page (see **Note**).

When responsive design was first introduced in the early '10s, there were only a handful of devices to worry about. So, we tended to base breakpoints on common device sizes (320 pixels for smartphones, 768 pixels for iPads, and so on), creating a dedicated layout for each breakpoint. It didn't take long until we had to deal with device widths at nearly every point from 240 to 3,000+ pixels. That device-based approach definitely didn't scale. These days, we let content be our guide.

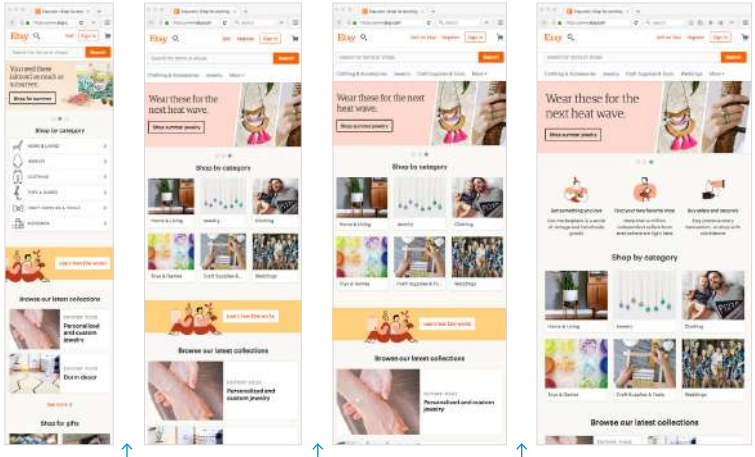
A breakpoint is a width at which you introduce a style change.

NOTE

*Visit **Etsy.com** today in your browser's responsive view, and you will find a simplified design with fewer breakpoints. I'm including an image of the 2018 version of the site because it provides such a good example of changing components between major page layout shifts, but it is also an example of how design best practices evolve over time.*

NOTE

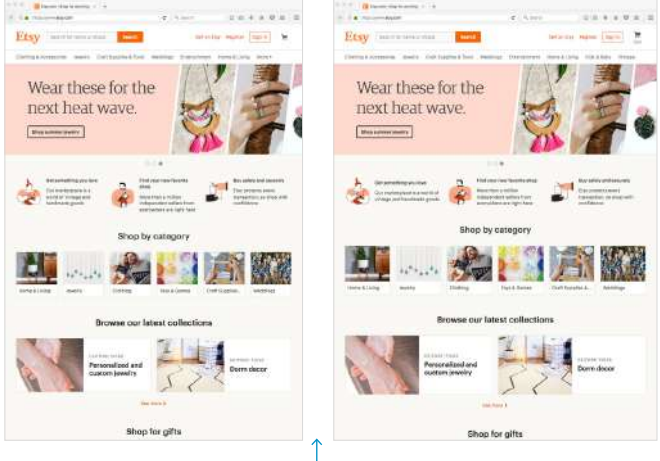
Because it's difficult to see the details in the screenshots, I've included a copy of FIGURE 19-16 *with the materials for this chapter (**learningwebdesign.com/6e/materials**).*



At the **480-pixel breakpoint**, the category navigation changes from a list to photos. "Register" is added to the top navigation bar.

At **501 pixels**, "Sell" becomes "Sell on Etsy" (a very subtle adjustment). You can also see more links in the navigation bar under the search field.

At **640 pixels**, the "How Etsy Works" images and messages move above the categories. In smaller views, they were accessible via the "Learn how Etsy works" link in a yellow bar.



At **901 pixels**, the search input form moves into the top header.

At **981 pixels**, the word "Cart" appears under the shopping cart icon. We now see the full list of navigation options in the header (no "More" link). At this point, the layout expands to fill larger windows until it reaches its maximum width of 1400 pixels. Then margins add space equally on the left and right to keep the layout centered.

FIGURE 19-16. A series of breakpoints used by Etsy's responsive site (circa 2018)

Content-based breakpoints

A better approach is to create breakpoints for the individual components of a page, rather than switching out the entire page at once (although for some sites that may work just fine). A common practice is to create the design for narrow screens first, and then resize the browser wider and pay attention to the point at which each part of the page starts to become unacceptable, as we did in EXERCISE 19-2. For example, your navigation might become too awkward and need a breakpoint at 500 pixels wide, but the one-column layout might be OK until it reaches 800 pixels, at which point a two-column design could be introduced.

It is common to create breakpoints for each component of the page rather than changing the entire page at once.

In his book *Responsive Design: Patterns & Principles* (self-published, ***ethanmarcotte.com***), Ethan Marcotte calls this "content out" design:

> *For me, that "content out" process begins by looking at the smallest version of a piece of content, then expanding that element until its seams begin to show and it starts to lose its shape. Once that happens, that's an opportunity to make a change—to introduce a breakpoint that reshapes the element and preserves its integrity.*

If you find that you have a lot of breakpoints within a few pixels or ems of one another, grouping them together may streamline your style sheet and process. And it doesn't hurt to keep the screen sizes of the most popular devices in mind, in case nudging your breakpoint up or down a little helps improve the experience for a whole class of users. To get a feel for the most common monitor and device screen sizes, you can check the current screen resolution statistics at StatCounter's Global Stats (***gs.statcounter.com/screen-resolution-stats***).

Component-based design

There has been a sea change in web design in the past decade or so, from designing entire pages to a more component- and pattern-based approach. Components (such as individual buttons) and patterns (a combination of components, such as a branded navigation bar with search) can be named, reused, and adapted for different contexts. One way to keep track of these bits and pieces and to ensure that they consistently and accurately represent the organization's brand is to use a design system. The **"Design Systems"** sidebar provides an introduction to this tool that has become central to the design and development workflow.

■ FURTHER READING

Brad Frost introduced "atomic design" in 2013 and established the terms "atoms, molecules, organisms, templates, pages" to describe the hierarchy of how smaller components get combined to form the pages we see in our browsers. Read his introductory article here: ***oreil.ly/cNHzX***. To learn more about this component-based approach to web development, read his ebook *Atomic Design* (self-published), available at ***atomicdesign.bradfrost.com***.

Content Hierarchy

Content is king on the web, so it is critical that content is carefully considered and organized before any code gets written. These are often tasks for information architects and content strategists who address the challenges of organizing, labeling, planning, and managing web content.

Design Systems

A **design system** is a set of reusable user interface (UI) elements, documentation, and guidelines that ensure a company's brand is consistent across all digital experiences. Over the last decade, they have emerged as an essential tool for ensuring consistency, efficiency, and scalability across digital projects. If you plan on working in web development professionally, it's nearly guaranteed that you will encounter a design system.

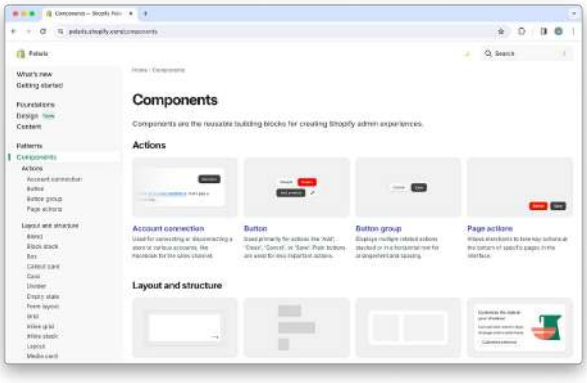
A design system establishes a common design language that designers and developers can use when building the interfaces of websites and native apps. It serves as the "single source of truth" when determining how the product should look and behave.

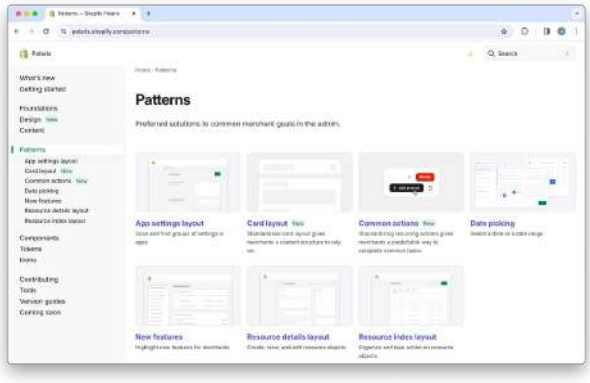
Design systems are the cornerstone of the modern component-driven design and development process. Beyond just documenting design decisions and code, they are a production tool for sharing and reusing UI components. Sites of all scales benefit from design systems, but they are critical for larger companies with products that extend into multiple channels (web, iOS, Android, newsletters, emails, and so on) and with large or even multiple distributed teams of designers and developers. Keeping all of the components, documentation, and principles in one place offers advantages such as:

- **Consistency.** When building and maintaining a strong brand, consistency is everything. With all components stored centrally, there is less guesswork when it comes to assembling a page. Reusable code snippets and UI elements ensure that elements like buttons look and work the same regardless of where they appear.
- **Time savings and efficiency.** Reusing precoded components saves time and effort, alleviating the need to create every component from scratch. Depending on how the system is implemented, you may be able change a key value (such as

Continued...

Shopify's Polaris Design System (*polaris.shopify.com*)





Salesforce's Lightning Design System (*www.lightningdesignsystem.com*)

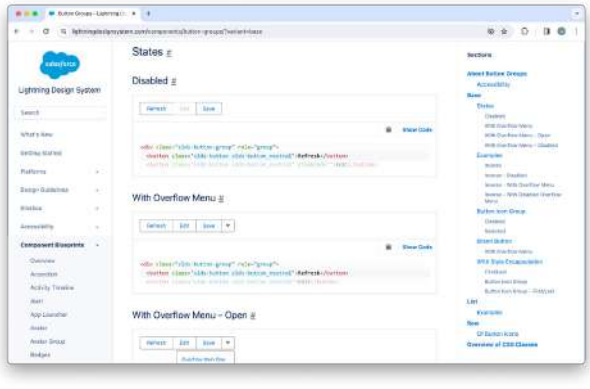


FIGURE 19-17. Small glimpses of the design systems for Shopify and Salesforce (2024)

Design Systems, *Continued*

the central brand color) in one place, and it will propagate to all the design/development tools across teams in an instant.

- **Cooperation.** A successful design system unifies teams within a company behind a single brand message and how it is manifested.
- **Onboarding.** The design system shortens the time it takes for new hires and contractors to get up to speed.

Some companies make their design systems publicly available, such as Shopify's Polaris Design System (*polaris.shopify.com*) and Salesforce's Lightning Design System (*lightningdesignsystem.com*), shown in FIGURE 19-17. I recommend that you browse through them and marvel at all the design decisions that go into creating products at that scale. It will also be evident how useful design systems are for keeping teams on track.

Component library

The core feature of a design system is a library of reusable UI components for buttons, form fields, icons, and other common elements. Components are stored as code snippets with documentation for how to use them. Rather than having to write HTML, CSS, and JavaScript from scratch every time, web designers and developers can implement the design system's components to help speed up their work and reduce errors. Components can be created and delivered in many technologies and platforms, like React, Angular, Web Components, Bootstrap, Tailwind CSS, and many more. For instance, a button Web Component might appear in the component library like this:

```
<ds-button variant="primary">Click me</ds-button>
```

In the development community, the term "design system" has become synonymous with the code component library. However, organizations may assemble additional documentation in an all-purpose design system ecosystem encompassing some or all of the following.

Pattern library

A **pattern** is an organization of multiple components that has a particular use—for example, a navigation bar (the pattern) may include a logo, links, and search form field components. A product card with an image, description, price, and call-to-action button is another example of a pattern. Some organizations make these more complex components available in a separate **pattern library**, but they may also be included in the main component library. See the article "The Art of Design System Recipes" (*oreil.ly/2QdIW*) by Brad Frost for strategies on dealing with multicomponent patterns.

UI tool library

A design system may also have a library of reusable UI components (buttons, form fields, icons, etc.) for use in design tools like Figma. Designers can grab the components from the central library for product screen designs, making their work more efficient and ensuring their designs are consistent with those produced by the rest of the team.

Design tokens

Complex design systems may take advantage of design tokens to maintain consistency in the design and development process. A **design token** assigns an intuitive name to a specific value that is used repetitively throughout a site (like a variable in a programming language).

Here's how that might work for an organization's main brand color (although any reused value can be tokenized). The primary brand color with the RGB value **#13B7BD** (a turquoise shade) is assigned the name "brand-color-primary". Designers and developers then apply the color "brand-color-primary" when the brand's color is called for, rather than using the specific hex value. If one day down the road, the company changes the RGB value of its main brand color, it can be changed in the token definition and the new value will be updated throughout the site and tools automatically. This is a very simplified description of design tokens, of course, but they are a powerful tool for large-scale design systems. Here are a few links with more in-depth information about design tokens:

- Stu Robson's "Awesome Design Tokens" page on GitHub (*oreil.ly/AnR7Q*)
- "Design Tokens: Beginner's Guide" by Nitish Kwaghal (*oreil.ly/BY_sz*)
- "Creating Themeable Design Systems" by Brad Frost (*oreil.ly/GaTDt*)

Style guide

A **style guide** documents all of the rules, standards, and assets that support an established brand identity. It generally includes logos, color palettes, typography, visual element styles, do's and don'ts for using brand assets, and more. It might also address voice and tone for written communications. The style guide may include general brand guidelines and principles, or those may be presented in a separate document. In addition to designers and developers, the style guide may be referenced by marketing teams and anyone else creating branded content for the organization.

Continued...

Design Systems, *Continued*

Other documentation

It is up to the organization to decide what is most useful to have documented for its teams and to determine what it is able to maintain. Here are just a few of the other types of documentation that you may see on a design system site:

- Brand guidelines
- Accessibility guidelines
- Documentation of processes and governance
- Information architecture
- Design system road map and releases

Hopefully, I've given you an understanding of design systems and their benefits. Of course, there is a lot more to be said on the topic than I can cover here, and you will find many excellent resources online. These should get you started:

- Design.Systems (*design.systems*) is a community hub where you can join the conversation about design systems and find out about relevant events.
- Figma, the company behind the hugely popular Figma design tool, publishes an introductory course on design systems: *oreil.ly/XMSZk*.
- The Component Gallery (*component.gallery*) is a collection of UI components from real-world design systems assembled by Iain Bean. It may help you prevent reinventing the wheel when it comes to designing elements like alerts, cards, checkboxes, drop-down menus, and more.
- Dan Mall provides a helpful introduction in his article and video "What Is a Design System? 6 Different Types of Design Systems" (*danmall.com/posts/what-is-a-design-system/*).

Organization and hierarchy across various views of the site are a primary concern, with a particular focus on the small-screen experience. It is best to start with an inventory of potential content and pare it down to what is most useful and important for all browsing experiences. Once you know what the content modules are, you can begin deciding in what order they appear on various screen sizes.

Keep in mind that you should strive for **content parity**—that is, the notion that the same content is accessible regardless of the device used to access the site. It might be that visitors need to follow a slightly different navigational path to that information, but dropping portions of your site on small screens because you think mobile users won't need them is false thinking. People do bop between devices mid-task, and you want to be sure they have everything they need.

Conditional Loading

Content parity doesn't mean that all of the content that fits on a large screen should be stuffed into the small-screen layout. All that scrolling and extra data to download isn't doing mobile users any favors.

A better approach is to use **conditional loading**, in which small-screen users get the most important content, with links to access supplemental content (comments, product details, ads, lists of links, etc.) when they want it. The information is available to them, just not all at once. Meanwhile, on larger screens, those supplemental pieces of content could be displayed automatically.

Conditional loading requires JavaScript to implement, so I won't be giving you the specific how-tos here, but it is good to know that there are alternatives to cramming every little thing onto every device.

This is a woefully brief introduction to what is perhaps the most important first step of creating a site, but it is just outside the focus of this book. To get up to speed properly with content strategy, particularly as it applies to responsive design, check out the following books:

- *Strategic Content Design: Tools and Research Techniques for Better UX* by Erica Jorgensen (Rosenfeld Media)
- *From Solo to Scaled: Building a Sustainable Content Strategy Practice* by Natalie Marie Dunbar (Rosenfeld Media)

Layout

Rearranging content into different layouts may be the first thing you think of when you picture responsive design, and with good reason. The layout helps form our first impression of a site's content and usability.

It's common for designers to create multiple designs for representative devices (small, medium, and large screens). These layouts are usually provided as major signposts, with the understanding that the details for in-between sizes will be worked out in the development process.

Many designers start the visual design process (establishing the look and feel, branding, general organization of the page, and so on) with the layout for the largest viewport size. More real estate allows more opportunities to develop visual themes and conventions. The smallest mobile layout can then be based on the visual themes and UI elements established in the large-screen design. Inevitably, solutions to problems in the small-screen layout will find their way back into the mid- and large-screen designs. Of course, design strategies vary from designer to designer, but in general, responsive design requires working on layouts for several device classes in parallel.

Responsive layout patterns

The manner in which a site transitions from a small-screen layout to a wide-screen layout must make sense for that particular site, but there are a few **layout patterns** (common and repeated approaches) that have emerged over the years. We can thank Luke Wroblewski (known for his "Mobile First" approach to web design) for doing a survey of how responsive sites handle layout. The article detailing his findings, "Multi-Device Layout Patterns" (*oreil.ly/vgfi_*), is getting on in years, but the patterns persist today. The following are the top patterns Luke named in his article (FIGURE 19-18):

Mostly fluid

This pattern uses a single-column layout for small screens and another fluid layout that covers medium and large screens, with a maximum width set to prevent it from becoming too wide. It generally requires less work than other solutions.

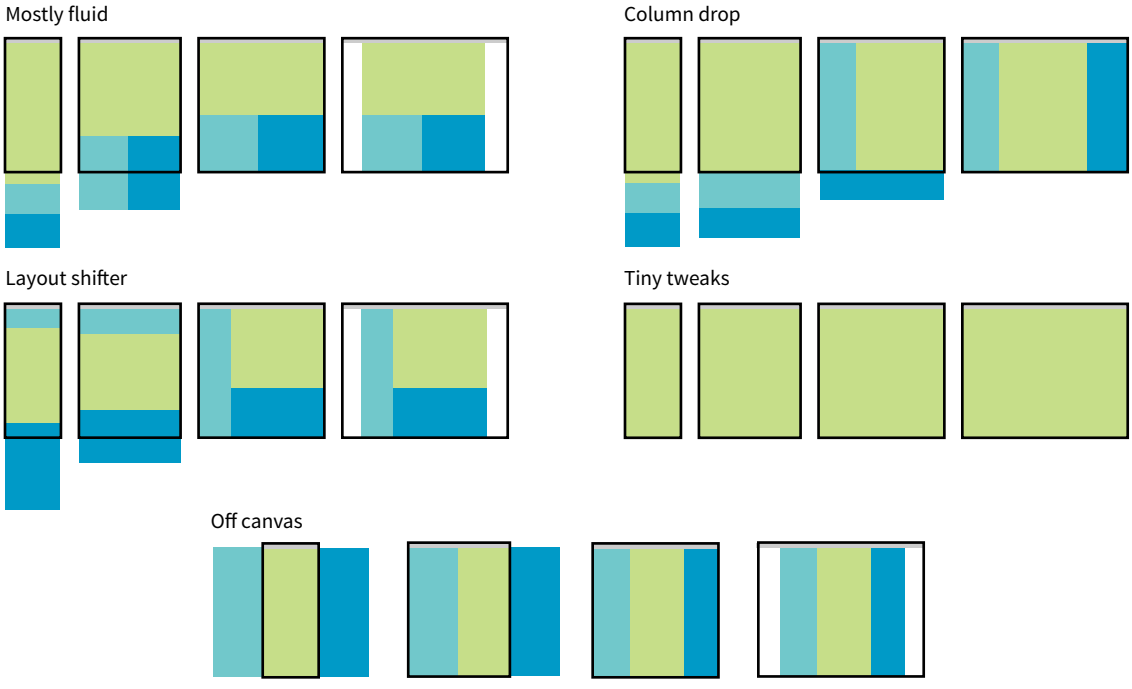


FIGURE 19-18. Examples of the responsive layout patterns identified by Luke Wroblewski

Column drop

This solution shifts between one-, two-, and three-column layouts based on available space. When there isn't room for extra columns, the sidebar columns drop below the other columns until everything is stacked vertically in the one-column view.

Layout shifter

If you want to get really fancy, you can completely reinvent the layout for a variety of screen sizes. Although expressive and potentially cool, this isn't necessary. In general, you can solve the problem of fitting your content to multiple environments without going overboard.

Tiny tweaks

Some sites use a single-column layout and make tweaks to type, spacing, and images to make it work across a range of device sizes.

Off canvas

As an alternative to stacking content vertically on small screens, you may choose to use an "off-canvas" solution. In this pattern, a page component is located just out of sight on the left or right of the screen and flies into view when requested. A bit of the main content screen remains visible on

the edge to orient users as to the relationship of the moving parts. This was originally made popular by Facebook, wherein Favorites and Settings were placed on a panel that slid in from the left when users clicked a menu icon.

You can see working examples of these and additional layout patterns on the Responsive Patterns page assembled by Brad Frost (*oreil.ly/gV05K*). More good examples are demonstrated in the article "5 Really Useful Responsive Web Design Patterns" by Joshua Johnson (*oreil.ly/ZjwC7*).

Responsive and Fluid Typography

Typography requires fine-tuning along the spectrum from small-screen to wide-screen views in order to keep it legible and attractive. When working out your designs across a range of screen sizes, pay attention to these typography details:

Font size

In general, the type size for headings and paragraphs should be larger on wide monitors than on handheld devices to take into account the typical distance from the user's eyes to the screen. Smaller (yet legible) text on small screens also means less scrolling. User interface text (like button labels and navigation) may need to be sized larger on large monitors as well. Font size values are best specified in rems, because they respond to the user's font size preferences and do not compound as ems do.

Line length

Varying viewport and component widths can wreak havoc on line lengths. You may find that you need to increase the font size of text elements for wider viewports or containers to maintain a comfortable line length of between 45 and 75 characters. See the **"Controlling Line Lengths with ch Units"** sidebar for another approach.

Line height

On average, line height should be about 1.5 (using a number value for the `line-height` property); however, slightly tighter line spacing (1.2 to 1.5) is easier to read with the shorter line lengths on small screens. Large screens, where the type is also likely to be larger, can handle more open line heights (1.4 to 1.8).

Padding and margins

On small screens, make the most of the available space by keeping left and right margins on the main column to a minimum. As screens get larger, you will likely need to increase side margins to keep the line lengths under control and just to add some welcome whitespace to the layout.

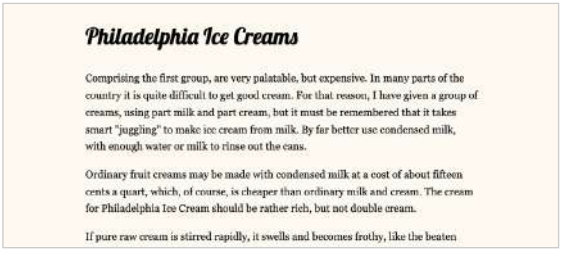
FIGURE 19–19 demonstrates a few of these pointers.

Controlling Line Lengths with ch Units

One way to ensure a text block will not be wider than 75 characters is to set its width using **ch** units. A **ch** unit is equal to the width of a 0 (zero) character in the current font. This example sets paragraphs to a width of 75ch:

```
p {
  width: 75ch;
}
```

Keep in mind that because the width is based on the 0 glyph, there may be slightly more than 75 characters if the line contains a lot of narrow characters (i, l) or slightly fewer if there are lots of wide characters, but it does get you in the ballpark. Also, because the width of a 0 in one font may be different from a 0 in another, **ch** unit lengths will vary by font.


Philadelphia Ice Creams
Comprising the first group, are very palatable, but expensive. In many parts of the country it is quite difficult to get good cream. For that reason, I have given a group of creams, using part milk and part cream, but it must be remembered that it takes smart "juggling" to make ice cream from milk. By far better use condensed milk, with enough water or milk to rinse out the cans.
Ordinary fruit creams may be made with condensed milk at a cost of about fifteen cents a quart, which, of course, is cheaper than ordinary milk and cream. The cream for Philadelphia Ice Cream should be rather rich, but not double cream.
If pure raw cream is stirred rapidly, it swells and becomes frothy, like the beaten

Narrow screens:
- Smaller type size
- Tighter line height
- Narrow margins

Wide screens:
- Larger type size
- Open line height
- Wider margins

FIGURE 19-19. General typography guidelines for small and large screens

Responsive typography

The traditional approach to resizing text based on screen size is to use a series of media queries and change the font size at each breakpoint (container queries could also be used). This is generally referred to as **responsive type**. In this example, the `h1` font size is 1.5rem when the viewport is on small screens, 1.8rem when the viewport reaches 600px, and 3rem when the viewport is at least 1200px wide:

```
h1 {
  font-size: 1.5rem;
}

@media (min-width: 600px) {
  h1 {
    font-size: 2rem;
  }
}
@media (min-width: 1200px) {
  h1 {
    font-size: 3rem;
  }
}
```

This is a fine solution, but it does have the drawback of abrupt shifts in font size at each breakpoint. When the viewport is 1,100 pixels wide, the headings will still be sized appropriately for the 600-pixel screen width.

Fluid typography

Fluid typography is a newer technique for making text resize smoothly by basing it on the size of the viewport or container. This approach takes care of proportional text resizing without using media queries at all! That means a lot less code and redundancy. Using the values from the previous example, with the fluid typography technique, when the viewport is 1,100 pixels wide, the `h1` will be approaching the maximum 3rem font size (FIGURE 19-20).

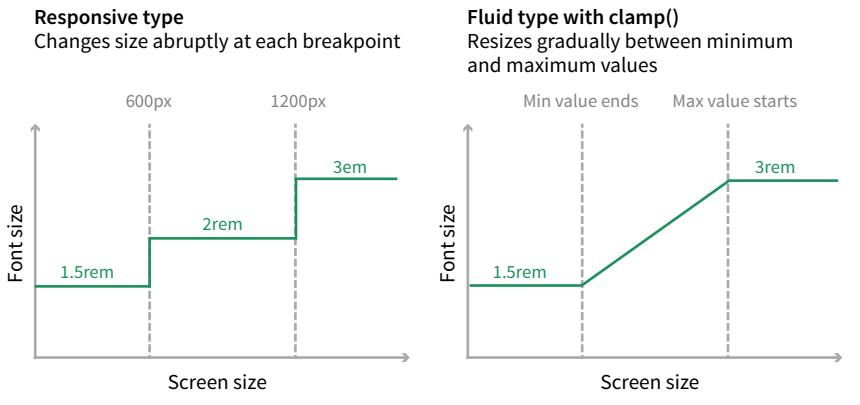


FIGURE 19-20. The difference between responsive and fluid type techniques

Fluid type values are appropriate for large display text, such as headlines, the big logo or message at the top of a home page, or any text that changes size significantly between handheld and desktop views. It's generally not necessary for text that only sizes up a few pixels, such as body text. In practice, fluid typography techniques are used along with responsive typography.

Fluid typography techniques are only a few years old, and developers have been trying out a variety of approaches, which will certainly show up in the results of a web search for "fluid typography." The most powerful and popular as of this writing is to use the **`clamp()`** function to set a minimum size, a preferred size, and a maximum size in one declaration. Compare the previous responsive code example to this example that sets a minimum size of 1.5rem, then gradually makes the text larger based on the viewport size, stopping at a maximum size of 3rem:

```
h1 {
  font-size: clamp(1.5rem, 0.955rem + 2.73vw, 3rem);
}
```

That's it! Set it and forget it! (Well, actually, more like set it, and test it, and test it, and test it.) The **`clamp()`** function takes three values, separated by commas. The first is the minimum font size (1.5rem). The third value is the maximum font size (3rem).

The second value, preferred size, bears more explanation. The preferred size is based on the viewport width, which makes the text get larger gradually as the viewport gets wider. However, because users can't zoom text set in **`vw`** units alone (an accessibility no-no), it is a best practice to supply the preferred size in rems, plus a "smattering" of **`vw`**.

Figuring out the preferred value **`rem`**/**`vw`** combination requires a bit of fancy math, but fortunately, there are several online tools that allow you to plug in values for minimum and maximum font sizes and viewport widths where the transition begins and ends. Here are a few:

CSS Math Functions

`clamp()` is one of four math functions that allow the browser to calculate specific numeric values. They can be used as values for any property that accepts a length, frequency, time, angle, percentage, or number value. Here's a quick look at the others:

`calc()`

Put a simple math problem (using +, ¸ –, *, or /) between the parentheses, and the resulting value will be applied to the property.
Example:
`width: calc(4em + 6px);`

`min()`

Provide a comma-separated list of values, and the smallest value will be applied to the property.
Example:
`width: min(20rem, 400px);`

`max()`

Provide a comma-separated list of values, and the largest value will be applied to the property.
Example:
`width: min(20rem, 400px);`

For more detailed explanations, I recommend the MDN Web Docs article "Using CSS Math Functions" (*oreil.ly/25c3ms*).

- **Min-Max-Value Interpolation** by 9Elements (*min-max-calculator.9elements.com*). This tool generates one clamp value at a time.
- **Fluid Type Scale Calculator** by Utopia (*utopia.fyi/type/calculator/*). This generates a set of variables that set text from small to xx-large, based on a numeric type scale, which is useful for setting up a type size system in one go.

We've been talking about basing the font size on the viewport width, but of course, you could change the font size based on the width of a container as well by using container queries and `cqi` units. This would be useful for headlines inside of flex or grid items that appear at different widths without a direct connection to the viewport.

It is also worth mentioning that other measurements that might change along with the viewport or container width, such as padding and margins, can be made fluid with `clamp()` values as well.

Accessibility issues

Using `vw` units in combination with rems solves the accessibility issue of allowing users to zoom text larger. Unfortunately, that solution doesn't go far enough in some scenarios. The Web Content Accessibility Guidelines (WCAG 2.1) define a rule (the "1.4.4 Resize text (AA)" rule, to be precise) that states that users must be able to resize their text 200% for better legibility. Because of the peculiarities of the `clamp()` formula, certain values prevent the text from reaching full 200% enlargement even at zoom levels of 500%.

As of this writing, there is not a straightforward solution; however, there are currently ongoing discussions with the CSS Working Group about adding a way to detect the zoom level to get around this shortcoming. Until that is implemented and supported by browsers, the best we can do is be careful with the values we feed the `clamp()` function and do a lot of testing. Maxwell Barvian did the math to target where the WCAG violations happen and came up with some general guidelines that you can read about in his article "Addressing Accessibility Concerns With Using Fluid Type" *(oreil.ly/zhZm4*). He also created the Accessible Fluid Type Generator, which alerts you when your values do not pass the 200% zoom test. It is available at *fluid.style*.

Browser support

The `clamp()` function has been supported by major browsers since 2020 (with 96% coverage as of this writing). If you need to support older browsers, you can set a static font size for the element as a fallback and use feature detection (discussed in **Chapter 21**) to serve responsive font sizes based with media queries to non-supporting browsers. You don't save any typing by including the full responsive styles in addition to the `clamp()` rule, but you will ensure a better typography experience for everyone.

FURTHER READING

The article "Modern Fluid Typography with CSS Clamp" by Adrian Bece provides a thorough introduction to fluid typography techniques (*oreil.ly/VYLsa*).

Cristian Diaz includes examples of container queries in his article "Fluid vs. Responsive Typography with CSS Clamp" (*oreil.ly/PdmQs*).

Navigation

Navigation feels a little like the Holy Grail of responsive web design. It is critical to get it right. Because navigation at desktop widths has pretty much been conquered, the real challenges come in re-creating navigation options on small screens. A number of successful patterns have emerged for small screens, which are briefly summarized here (FIGURE 19-21):

Top navigation

If your site has just a few navigation links, they may fit just fine in one or two rows at the top of the screen.

Priority +

In this pattern, the most important navigation links appear in a line across the top of the screen alongside a "More" link that exposes additional options. The pros are that the primary links are in plain view, and the number of links shown can increase as the device width increases. The cons include the difficulty of determining which links are worthy of the prime small-screen real estate.

Select menu

For a medium list of links, some sites use a `select` input form element. Tapping the menu opens the list of options using the select menu UI of the operating system. The advantage is that it is compact, but on the downside, forms aren't typically used for navigation, and the menu may be overlooked.

Link to footer menu

One straightforward approach places a Menu link at the top of the page that links to the full navigation, located at the bottom of the page. The risk with this pattern is that it may be disorienting to users who suddenly find themselves at the bottom of the scroll.

Accordion sub-navigation

When there are a lot of navigation choices with sub-navigation menus, the small-screen solution becomes more challenging, particularly when you can't hover to get more options as you can with a mouse. Accordions that expand when you tap a small arrow icon are commonly used to reveal and hide sub-navigation. They may even be nested several levels deep. Some tips for multiple nested accordions include using color and typography to make each level visually distinct, not using too many styles of icons ("open" and "closed" arrows may be all you need), and making sure that links are styled to look like links.

To avoid nesting navigation in accordion submenus, some sites simply link to separate landing pages that contain a list of the sub-navigation options for that section.

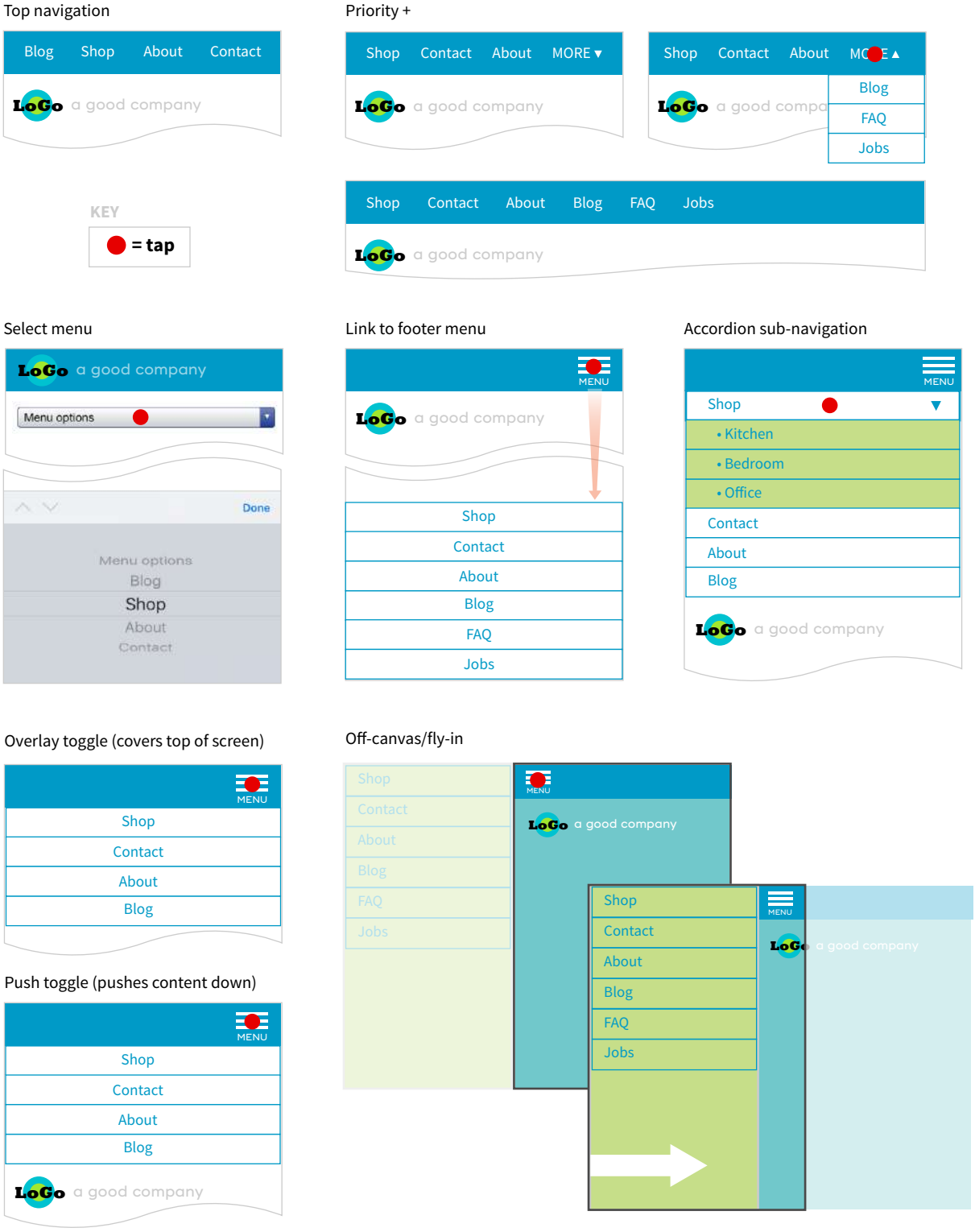


FIGURE 19-21. Responsive navigation patterns

Push and overlay toggles

In toggle navigation, the navigation is hidden but expands downward when the menu link is tapped. It may push the main content down below it (push toggle) or slide down in front of the content (overlay toggle).

Off-canvas/fly-in

This popular pattern puts the navigation in an off-screen panel to the left or right of the main content that slides into view when you tap the menu icon. The advantage to this method is that it affords more space for a lot of navigation options. The disadvantage is that the animation effects can be distracting or annoying, and it doesn't offer much that can't be accomplished with accordions.

For a deeper dive into the pros and cons of various navigation patterns, read Brad Frost's articles "Responsive Navigation Patterns" (**oreil.ly/X4G0n**) and "Complex Navigation Patterns for Responsive Design" (**oreil.ly/X-435**). Brad also includes examples of these patterns and more on his Responsive Patterns page (**oreil.ly/Y6-vp**).

For excellent detailed examples of several complex navigation options, see the article "Designing Navigation for Mobile: Design Patterns and Best Practices" by Vitaly Friedman on *Smashing Magazine* (**oreil.ly/VCgrW**).

Images

Images require special attention in responsive designs. Here is a quick rundown of some of the key recommendations, most of which should sound familiar:

- Use responsive image markup techniques to provide multiple versions of key images for various sizes and resolutions. See my article "Responsive Image Markup," available at **learningwebdesign.com/articles**, for more information.
- Serve the smallest version as the default to prevent unnecessary data downloads.
- Be sure that important image detail is not lost at smaller sizes. Consider substituting a cropped version of the image for small screens.
- Avoid putting text in graphics, but if it is necessary, provide alternate versions with larger text for small screens.

Designing for Fingers

Keep in mind that people use their fingers to get around on their devices, which these days include smartphones, tablets, and even desktop-sized screens.

Links in navigation should be big enough to easily target with thumbs and fingertips. Apple requires a "hit region" of 44 pixels for buttons in its apps, and that's a good ballpark to keep in mind for links on web pages as well.

Another consideration for touch devices is that there is no hover state. Hovering has become the convention for opening sub-navigation on web pages on the desktop, but with no mouse, that experience is very different. One solution is to deliver styles that do not require hovering to devices that lack that ability using media queries that test for the hover and/or pointer features. The article "A Guide to Hover and Pointer Media Queries" by Christian Diaz (**oreil.ly/CElqB**) does a thorough job of explaining how they work.

A really great book about all of this stuff is Josh Clark's *Designing for Touch*, which you can purchase at **oreil.ly/kRr9D**.

Videos

Videos, like images, need to scale up and down while maintaining their aspect ratio in a responsive environment, but the **max-width** trick that works for images does not apply to videos. The current best practice for making videos resize proportionally is to set the **width** and **height** of the **video** element to 100% and use the **aspect-ratio** property to restrict the ratio to 16:9 (or whatever the aspect ratio of the video might be):

```
video {
  width: 100%;
  height: 100%;
  aspect-ratio: 16 / 9;
}
```

For videos embedded with an **iframe**, apply **aspect-ratio** to the containing **iframe**:

```
iframe {
  width: 100%;
  height: 100%;
  aspect-ratio: 16 / 9;
}
```

The **aspect-ratio** property has been supported by all major browsers since 2021, but if you need to support old browser versions or Internet Explorer, you can use an old padding hack created by Thierry Koblentz and described in his article "Creating Intrinsic Ratios for Video" (*oreil.ly/0ile3*).

There is also responsive markup for videos that allows you to serve appropriately sized video files to devices using the **media** attribute to test for screen size. Scott Jehl, who was instrumental in getting this feature into the current HTML spec, provides a good explanation in his article "How to Use Responsive HTML Video (...and Audio!)" (*scottjehl.com/posts/using-responsive-video/*).

Finally, if you want to display a video in the background of a responsive layout element, see the step-by-step tutorial "How to Create a Responsive Video Background in HTML and CSS" by Adeyele Paul (*oreil.ly/wUlUG*).

Special Content

Without the luxury of wide-open desktop browser windows, some common page elements pose challenges when it comes to fitting on smaller screens:

Forms

Forms often take a little finagling to fit the available space appropriately. Flexbox is a great tool for adding flexibility and conditional wrapping to form fields and their labels. A web search will turn up some fine tutorials. Also make sure that your form is as efficient as possible, with no unnecessary fields (this is good advice for any screen size). Finally, consider that

form inputs will be used with fingertips, not mouse pointers, so increase the target size by adding ample padding or margins and by making labels tappable to select an input.

Interactive elements

A big embedded map may be great on a desktop view of a site, but it is less useful when it is the size of a postage stamp. Consider whether some interactive features should be substituted with other methods for performing the same task. In the case of a map, adding a link to the map can trigger opening the device's native mapping app, which may provide a better small-screen experience. Other interactive components, such as carousels, can be adapted for smaller viewports. The `aspect-ratio` property may be useful for resizing other embedded media proportionally as well.

Tables

One of the greatest challenges in small-screen design is how to deal with large data tables. Not surprisingly, because there are many types of tables, there are also many solutions. See the sidebar **"The Trouble with Tables"** for more information and resources.

The Trouble with Tables

Large tables, such as those shown back in FIGURE 8-1, can be difficult to use on small-screen devices. By default, they are shrunk to fit the screen width, rendering the text in the cells too small to be read. Users can zoom in to read the contents, but then only a few cells may be visible at a time, and it is difficult to parse the organization of headings and columns.

Designers and developers have created a number of approaches for making tables responsive. One option is using flexible values that allow the table to squish into smaller spaces. If that's not enough, there are solutions that involve some advanced web development mojo (more than we can take on here). Responsive table tools are also built into some frameworks, like Bootstrap, and there are plug-ins available for WordPress, so you may not need to code it alone.

There are three general approaches for making tables responsive: scrolling, stacking, and hiding. Here's a quick rundown:

- For scrolling solutions, the table stays as wide as it needs to be, and users can scroll to the right to see the columns that won't fit. This can be accomplished with JavaScript or CSS alone. You can even anchor the left column to the window so that it stays put when the rest of the table scrolls.
- Another approach is to stack up the entries in a long, narrow scroll. Each entry repeats the headers, so the data is always presented with the proper context. Again, you could do this with JavaScript or CSS. The downsides are that the list can end up being very long and it makes it difficult to compare entries, but at least all of the information is visible without horizontal scrolling.
- You may also choose to hide certain columns of information when the page first loads on small devices and give the user the option to click to see the whole table or to toggle specific columns on and off. That's a little riskier from an interaction design perspective; those columns might just not be seen at all.

The best approach depends on the type of data you're publishing and how the table is expected to be used. If you are interested in learning more, I recommend the following resources:

- "Accessible Front-End Patterns for Responsive Tables, Part I" by Adrian Bece on *Smashing Magazine* (*oreil.ly/Q7Dry/*)
- "Accessible Front-End Patterns for Responsive Tables, Part II" by Adrian Bece on *Smashing Magazine* (*oreil.ly/PFZ44*)
- "Accessible, Simple, Responsive Tables" by Davide Rizzo on *CSS-Tricks* (*oreil.ly/5xBcO/*

A FEW WORDS ABOUT TESTING

In the EXERCISE 19-2, we relied on the responsive view in Chrome's developer tools to make decisions about style changes at various sizes. But although it's a handy tool for creating an initial design, much more testing is required before the design can be considered ready for final launch. That is even more critical for sites that include features that rely on JavaScript or server-side functionality.

There are three general categories of site testing environments: real devices, virtual devices, and third-party services (which use a combination of real and virtual devices on your behalf).

Real Devices

Testing a site on a variety of real devices and operating systems gives you the most accurate feel for the user experience. Beyond just seeing how the site looks, testing on real devices shows you how your site *performs*. How fast does it load? Are the links easy to tap? Do all the interactive features work smoothly? Do they work at all?

Ideally, a site should be tested on a variety of browsers on an iPhone, Android phone, iPad and other tablets, and computers running macOS, Microsoft Windows, and Linux. Some companies have an in-house collection of test devices (often referred to as a **device lab**) specifically for testing. It's common to use live reloading software such as Browsersync (***browsersync.io***) to push the site to all of the devices simultaneously. A web search will turn up additional options for testing on multiple devices at once.

If you are building small sites on your own and don't have access to a big device lab, simply testing your site on friends' devices or paying a quick visit to a store that sells phones and tablets can provide valuable insight.

Virtual Devices

Because device labs are expensive and labor-intensive to build and maintain, developers may choose to test sites on **virtual devices**. A virtual device is software that mimics the features and behaviors of an actual smartphone or mobile device. There are two types of virtual devices: emulators and simulators.

An **emulator** provides the closest experience to testing on a real device, incorporating hardware features such as sensors, geolocation, battery state, and so on. They are a good choice if you are testing how your app or software interacts with the hardware of the device. The downside is that they require a lot of processing power and are generally slower to use. One example is the emulator Android Studio, which runs on your own machine (Windows, macOS, or Linux) and allows you to test on Android devices.

A **simulator** mimics the software layer of the device only and generally works faster, although it is not able to test hardware features. The iOS SDK (software development kit) includes an iOS simulator that runs on macOS only. iOS simulators for Windows can be turned up with a web search (see **Note**). An easy way to try out a simulator is to install the Mobile FIRST plug-in for the Chrome browser (***webmobilefirst.com***), which lets you see what your site looks like on a small collection of current devices, as shown in FIGURE 19-22. It is free to use, but there is a paid PRO version as well.

NOTE

Because the chips and code for Apple devices are proprietary, true emulators are very difficult to come by; simulators are usually the closest you will get.

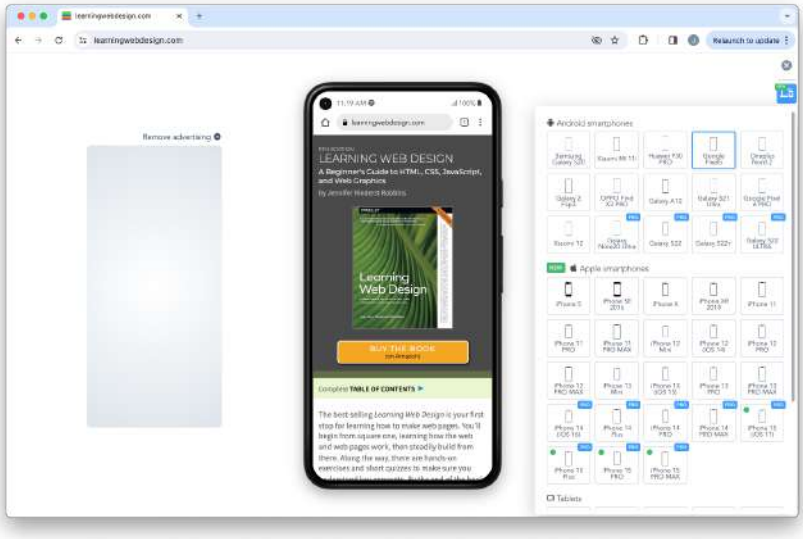


FIGURE 19-22. The Mobile FIRST Chrome extension simulating a Google Pixel 5 smartphone

Third-Party Services

Although it is possible to have a spread of physical devices as well as virtual devices on the computer you use for development, it is far more common for product teams to take advantage of cross-browser testing services. These services provide access to thousands of combinations of devices, browsers, and operating systems and can conduct automatic testing and provide bug reports. They charge a monthly fee, but most have free trial periods so you can try them out on your own sites. There are many cross-browser testing services out there, but here are a few that are popular as of this writing:

- BrowserStack (***browserstack.com***)
- Perfecto (***perfecto.io***)
- BitBar (***smartbear.com/product/bitbar***)

FIGURE 19-23 shows how the ***learningwebdesign.com*** site (circa 2024) looks on BrowserStack Live (***browserstack.com/live***).

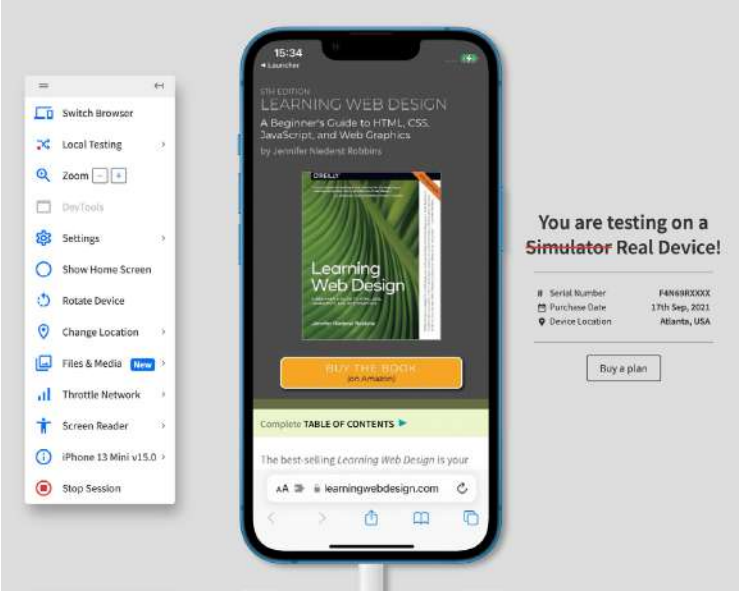


FIGURE 19-23. Device simulation on BrowserStack Live

MORE RWD RESOURCES

We've covered the mechanics of using fluid layouts, flexible images, and media queries to make a page that is usable across a wide range of screen sizes. We've looked at the design concerns and some common responsive patterns for layout, navigation, typography, and images. You even got a chance to try out creating a responsive page on your own. But this is really only the tip of the iceberg, and I encourage you to continue learning about responsive design, particularly if you are considering web design or development as a career. Following is a list of resources that I've found helpful and which should point you in the right direction.

Books

Responsive Web Design with HTML5 and CSS*, 4th ed.*, by Ben Frain (**Packt Publishing**)

> This is a very thorough overview of web design techniques that is a bit more advanced technically than the book you are currently reading.

Smashing Book #5: Real-Life Responsive Web Design, various authors (*Smashing Magazine*)

> A collection of practical techniques and strategies from prominent web designers.

Responsive Web Design**, 2nd ed.*, by Ethan Marcotte (ethanmarcotte.com/books/responsive-web-design/***)

This book is important for historical purposes; plus, it's just plain fun to read.

Online Resources

***Responsive Resources** (**oreil.ly/SUiwc**)*

For one-stop shopping for everything you could possibly want to know about RWD, look no further than Brad Frost's Responsive Resources page. He has gathered hundreds of links to resources related to strategy, design tools, layout, media queries, typography, images, components, development, testing, content management systems, email, tutorials, and more. Seriously, there is enough here to keep you busy for months.

***Frontend Dogma** (**oreil.ly/6aHCv**)*

This site, published by Jens Oliver Meiert, posts news and information for web developers and keeps an up-to-date list of links to useful responsive design resources.

More Inspiration

You can get lost in all the cool designs, the vast majority of which are responsive, on these web design showcase sites:

- Awwwards (***awwwards.com/websites/***)
- Siteinspire (***siteinspire.com***)
- CSS Winner (***csswinner.com***)
- Best Website Gallery (***bestwebsite.gallery***)

TEST YOURSELF

Here we are at the end of another chapter. You know what that means...*quiz time!* Check the answers in **Appendix A** if you're stumped.

1. What does this do?

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

2. Write a rule that makes images get smaller when their containers get smaller in the layout.

3. Write a rule that sets the margin on the left and right side of the page to 10% when the viewport is 60em or wider.

4. List three ways in which a media query may be implemented.

5. What does this do?

```
h1 {
  margin: clamp(0.375rem, -0.17rem + 2.73vw, 1.875rem);
}
```

6. Describe two approaches to presenting navigation with submenus on small screens.

7. Write the rule to make the "wrapper" **`div`** into a containing context for container queries.

```
<div id="wrapper">
  <h1>Muffins</h2>
  <ul>
    <li>Blueberry</li>
    <li>Chocolate chip</li>
    <li>Banana</li>
  </ul>
</div>
```

8. Using the code example in question 7, write a rule that makes the **`ul`** into a flex container when the "wrapper" **`div`** is 800px wide or wider.

9. List three options for testing websites on multiple devices.

10. What is the difference between a style guide and a design system?

11. What is the difference between a component and a pattern?

12. What is the difference between responsive and fluid typography?

CHAPTER
20

TRANSITIONS, TRANSFORMS, AND ANIMATION

IN THIS CHAPTER

Creating smooth transitions

Moving, rotating, and scaling elements

Combining transitions and transforms

Keyframe animations

In earlier chapters, you've seen CSS used for visual effects like rounded corners, color gradients, and drop shadows that previously had to be created with graphics. In this chapter, we'll look at some properties for producing animated interactive effects that were previously possible only with JavaScript or animation tools like Flash (now obsolete).

We'll start with CSS Transitions, a nifty way to make style changes fade smoothly from one to another. Then we'll discuss CSS Transforms for repositioning, scaling, rotating, and skewing elements and look at how you can animate those transformations. I'll close out the chapter with brief introduction to CSS Animations, which are important to know about but too vast to cover fully here, so I'll give you just a taste.

The problem with this chapter is that animations and time-based effects don't work on paper, so I can't show them off here. I did the next best thing, though, and made the source code for the figures available with the materials for this chapter (***learningwebdesign.com/6e/materials***) in a folder called *figures*. Just open the file in your browser.

EASE-Y DOES IT (CSS TRANSITIONS)

Picture, if you will, a link in a navigation menu whose background changes from blue to red when the mouse hovers over it. The background is blue... mouse passes over it...BAM! Red! It goes from state to state instantly, with no states in between. Now imagine putting your mouse over the link and having the background gradually change from blue to red, passing through several shades of purple on the way. It's smoooooth. And when you remove the mouse, it fades back down to blue again.

NOTE

This section focuses on the CSS Transitions Level 1 (w3.org/TR/css-transitions-1/) spec, which has been well supported since 2013. The Level 2 spec (w3.org/TR/css-transitions-2/) is currently a Public Working Draft; it includes an additional property, `transition-behavior`*, that restricts transitions for some properties and an at-rule,* `@starting-style`*, that starts a transition when an element is rendered. Additionally, Level 2 defines transition animations more clearly and consistently with the Web Animation API. These new features are not well supported as of this writing.*

That's what CSS Transitions (see **Note**) do. They smooth out otherwise abrupt changes to property values between two states over time by filling in the frames in between. Animators call that **tweening**. When used with reserve, CSS Transitions can add sophistication and polish to your interfaces and even improve usability.

Transition Basics

Transitions are a lot of fun, so let's give them a whirl. When applying a transition, you have a few decisions to make, each of which is set with a `transition-*` property:

- Which CSS property to change (`transition-property`) *(required)*
- How long it should take (`transition-duration`) *(required)*
- The manner in which the transition accelerates (`transition-timing-function`)
- Whether there should be a pause before it starts (`transition-delay`)

Transitions require a beginning state and an end state. The element as it appears when it first loads is the beginning state. The end state needs to be triggered by a state change such as `:hover`, `:focus`, or `:active`, which are the ones I'll be using for the examples in this chapter. You could use JavaScript to change the element (such as adding a `class` attribute) and use that as a transition trigger as well.

Let's put all of this together with a simple example. FIGURE 20-1 visualizes that blue-to-red link background transition you imagined earlier.

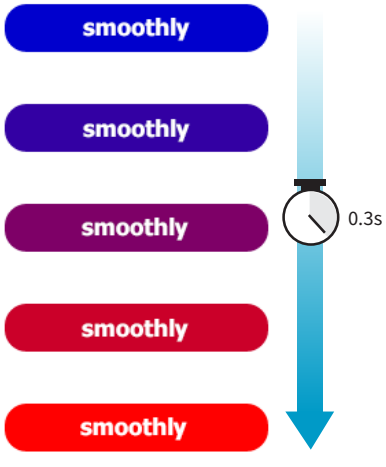


FIGURE 20-1. The background color gradually fades from blue to red over .3 seconds when a transition is applied.

The transition properties are applied to the object that will be transitioned—in this case, the `a` element in its normal state. You'll see them in the block of other style declarations for `.smooth`, including the starting background color ("mediumblue"). For the end state, I've changed the background color of the link to "red" by declaring the `background-color` for the `:hover` state (and `:focus` too, in case someone is tabbing through links with a keyboard). With the transition properties in place, it takes .3 seconds to get to the red background color on hover or focus. There's nothing special about the markup. I added a `class` so I could be specific about which links receive transitions.

THE MARKUP

```
<a href="..." class="smooth">smoothly</a>
```

THE STYLES

```
.smooth {
  ...
  transition-property: background-color;
  transition-duration: .3s;
}
.smooth:hover, .smooth:focus {
  background-color: red;
}
```

Specifying the property

transition-property

Values: *property-name* | `all` | none

Default: `all`

Applies to: All elements, `::before` and `::after` pseudo-elements

Inherits: No

`transition-property` identifies the CSS property that is changing and that you want to transition smoothly. In our example, it's `background-color`. You can also change the foreground color, borders, dimensions, and font- and text-related attributes. In fact, most CSS properties are animatable, and those that are not (like the animation properties themselves) wouldn't make sense to animate anyway. The general rule is that if its value is a color, length, or number, that property can be a transition property.

Add the transition properties to the object that will be transitioned.

Specifying how much time it should take

transition-duration

Values: *time*

Default: `0s`

Applies to: All elements, `::before` and `::after` pseudo-elements

Inherits: No

`transition-duration` sets the amount of time it takes for the animation to complete in seconds (`s`) or milliseconds (`ms`). I've chosen .3 seconds, which is just enough to notice that something happened but not so long that the transition feels sluggish or slows the user down. There is no correct duration, of course, but .2s seems to be a popular transition time for UI elements and is a good place to start. Experiment to find the duration that makes sense for your application.

Timing Functions

transition-timing-function

Values: `ease` | `linear` | `ease-in` | `ease-out` | `ease-in-out` | `step-start` | `step-end` | `steps` | `cubic-bezier(#,#,#,#)`

Default: `ease`

Applies to: All elements, `::before` and `::after` pseudo-elements

Inherits: No

The property and the duration are required and form the foundation of a transition, but you can refine it further. There are a number of ways a transition can roll out over time. For example, it could start out fast and then

slow down, start out slow and speed up, or stay the same speed all the way through, just to name a few possibilities. You can think of this as the transition "style," but in the spec, it is known as the **timing function** or **easing function**.

The timing function you choose has a big impact on the feel and believability of animations, so if you plan on using transitions and CSS animations, it's a good idea to get familiar with the options.

In this example, the value of `transition-timing-function` is set to `ease-in-out`, which makes the transition start out slow, then speed up, then slow down again as it comes to the end state:

```
.smooth {
  ...
  transition-property: background-color;
  transition-duration: .3s;
  transition-timing-function: ease-in-out;
}
```

The `transition-timing-function` property takes one of the following keyword values:

`ease`

Starts slowly, accelerates quickly, and then slows down at the end. This is the default value and works just fine for most short transitions.

`linear`

Stays consistent from the transition's beginning to end. Because it is so consistent, some say it has a mechanical feeling.

`ease-in`

Starts slowly, then speeds up.

`ease-out`

Starts out fast, then slows down.

`ease-in-out`

Starts slowly, speeds up, and then slows down again at the very end. Similar to `ease`, but with less pronounced acceleration in the middle.

`cubic-bezier(`*x1*`,`*y1*`,`*x2*`,`*y2*`)`

The acceleration of a transition can be plotted with a curve called a **Bezier curve**. The steep parts of the curve indicate a fast rate of change, and the flat parts indicate a slow rate of change. FIGURE 20-2 shows the Bezier curves that represent the function keywords as well as a custom curve I created myself. You can see that the `ease` curve is a tiny bit flat in the beginning, gets very steep (fast), then ends flat (slow). The `linear` keyword, on the other hand, moves at a consistent rate for the whole transition.

You can get the feel of your animation *just right* by creating a custom curve. Lea Verou's ***Cubic-Bezier.com*** is a neat tool for playing around with transition timing and generating the resulting code. The four numbers in the value represent the *x* and *y* positions of the start and end Bezier curve handles (the pink and blue dots in FIGURE 20-2). Bezier curve editors are also available in the developer tools for Chrome and Firefox.

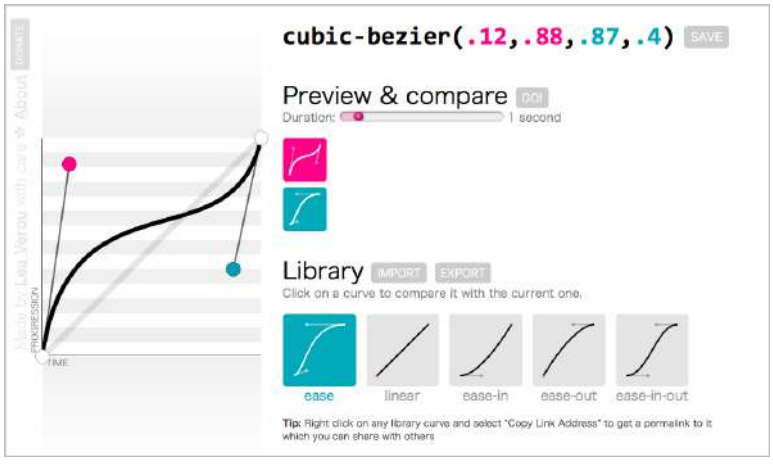


FIGURE 20-2. Examples of Bezier curves from Cubic-Bezier.com. On the left is my custom curve that starts fast, slows down, and ends fast.

`steps(#, start|end)`

Divides the transitions into a number of steps, as defined by a stepping function. The first value is the number of steps, and the **`start`** and **`end`** keywords define whether the change in state happens at the beginning or end of each step. Step animation is especially useful for keyframe animations with sprite images. For a better explanation and examples, I recommend the article "Using Multi-Step Animations and Transitions" by Geoff Graham on CSS-Tricks (***oreil.ly/zFU6F***).

`step-start`

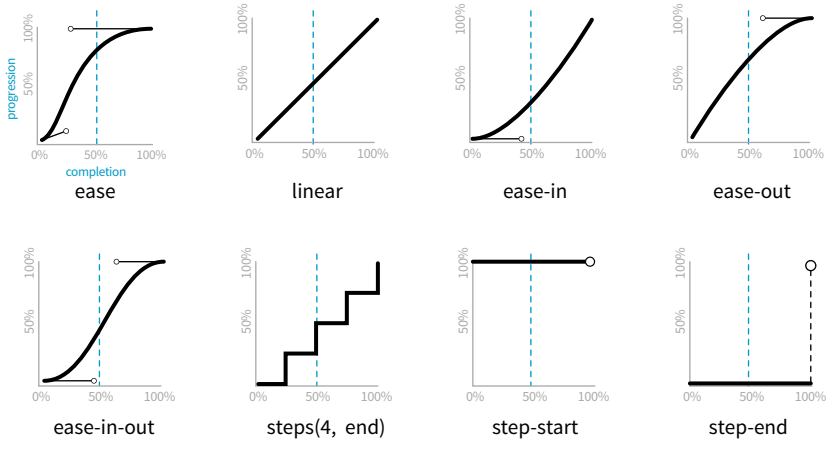
Changes states in one step, at the beginning of the duration time (the same as **`steps(1, start)`**). The result is a sudden state change, the same as if no transition had been applied at all.

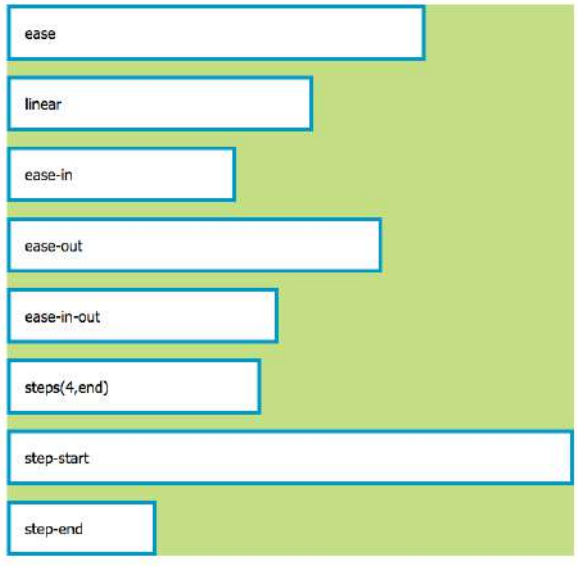
`step-end`

Changes states in one step, at the end of the duration time (the same as **`steps(1, end)`**).

It's difficult to show the various options on a still page, but I have put together a little demo, which is illustrated in FIGURE 20-3 and available in the *figures* folder with the materials for this chapter. The width of each labeled element

(white with a blue border) transitions over the course of 4 seconds when you hover over the green box. They all arrive at their full width at exactly the same time, but they get there in different manners. The image shown in FIGURE 20-3 was taken at the 2-second mark, halfway through the duration of the transition.





The width of the white boxes is set to transition from 0 to 100% width over 4 seconds. This screenshot shows the progress after 2 seconds (50%) for each timing function.

FIGURE 20-3. In this `transition-timing-function` demo, the elements reach full width at the same time but vary in the manner in which they get there. If you'd like to see it in action, the *figures.html* file is available with the materials for this chapter.

Setting a Delay

transition-delay

Values: *time*

Default: 0s

Applies to: All elements, `:before` and `:after` pseudo-elements

Inherits: No

The **`transition-delay`** property, as you might guess, delays the start of the animation by a specified amount of time. In the following example, the background color transition starts .2 seconds after the pointer moves over the link:

```
.smooth {
  ...
  transition-property: background-color;
  transition-duration: .3s;
  transition-timing-function: ease-in-out;
  transition-delay: .2s;
}
```

The Shorthand transition Property

Thankfully, the authors of the CSS Transitions spec had the good sense to give us the shorthand **`transition`** property to combine all of these properties into one declaration. Here is the syntax:

```
transition: property duration timing-function delay;
```

The values for each of the **`transition-*`** properties are listed out, separated by character spaces. The order isn't important as long as the duration (which is required) appears before the delay (which is optional). If you provide only one time value, it will be assumed to be the duration.

Using the blue-to-red link example, I could combine the four transition properties I've applied so far into this one line:

```
.smooth {
  ...
  transition: background-color .3s ease-in-out .2s;
}
```

Definitely an improvement.

Applying Multiple Transitions

So far, I've changed only one property at a time, but it is possible to transition several properties at once. Let's go back to the "smoothly" link example. This time, in addition to changing the background from blue to red, I'd like the **`letter-spacing`** to increase a bit. I also want the text color to change from white to black, but more slowly than the other animations. FIGURE 20-4 attempts to show these transitions on this static page.

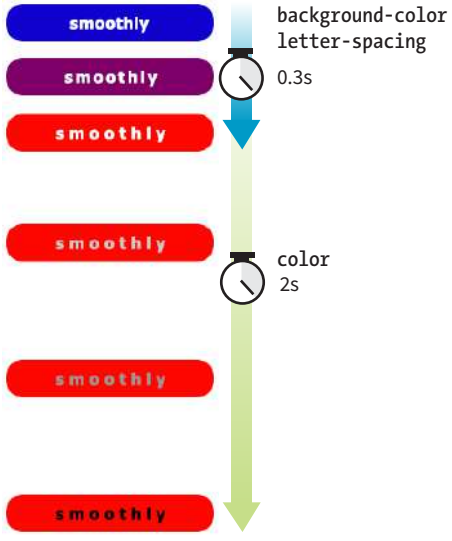


FIGURE 20-4. The `color`, `background-color`, and `letter-spacing` change at different paces.

One way to do this is to list all of the values for each property separated by commas, as shown in this example:

```
.smooth {
  ...
  transition-property: background-color, color, letter-spacing;
  transition-duration: .3s, 2s, .3s;
  transition-timing-function: ease-out, ease-in, ease-out;

}
.smooth:hover, .smooth:focus {
  background-color: red;
  letter-spacing: 3px;
  color: black;
}
```

The values are matched up according to their positions in the list. For example, the transition on the **`color`** property (second in the list) has a duration of 2s and uses the **`ease-in`** timing function. If one list has fewer values than the others, the browser repeats the values in the list, starting over at the beginning.

You can use multiple values for the shorthand **`transition`** property as well. The same set of styles we just saw could also be written as follows (I stacked them to make them easier to follow, but that is not required):

```
.smooth {
  ...
  transition: background-color .3s ease-out,
              color 2s ease-in,
              letter-spacing .3s ease-out;
}
```

A Transition for All Occasions

But what if you just want to add a little bit of smoothness to all your state changes, regardless of which property might change? For cases where you want the same duration, timing function, and delay to apply to all transitions that might occur on an element, use the **all** value for **transition-property**. In the following example, I've specified that any property that might change for the **.smooth** element should last .2 seconds and animate via the **ease-in-out** function:

```
.smooth {
  ...
  transition: all .2s ease-in-out;
}
```

For user interface changes, a short, subtle transition is often all you need for all your transitions, so the **all** value will come in handy. That wraps up our lesson on CSS Transitions. Now you give it a try in EXERCISE 20-1.

EXERCISE 20-1. Trying out transitions

In this exercise, we're going to create the rollover and active states for a menu link (FIGURE 20-5) with animated transitions. I've put together a starter document (*transitions.html*) for you; you'll find it with the materials for this chapter at *learningwebdesign.com/6e/materials*.

FIGURE 20-5. In this exercise, we'll create transitions between these link states.

Take a look at the styles that are already applied. The list has been converted to a horizontal menu with Flexbox. The **a** element has been set to display as a block element, underlines are turned off, dimensions and padding are applied, and the color, background color, and border are established. I used the **box-shadow** property to make it look as though the links are floating off the page.

NOTE

If you're using a touch device for this exercise, you'll miss out on this effect because there is no **:hover** *state on touchscreens. You may see the* **:hover** *state with a single tap. Transitions triggered by a click/tap or when the page loads will work on all devices, but they are not covered here.*

EXERCISE 20-1. Continued

1. First, we'll define the styles for the **:hover** and **:focus** states. When the user puts the pointer over or tabs to a link, make the background color change to green (**#c6de89**) and the border color change to a darker shade of green (**#a3c058**):

```
a:hover, a:focus {
  background-color: #c6de89;
  border-color: #a3c058;
}
```

2. While the user clicks the link (**:active**), make it move down by 3 pixels as though it is being pressed. Do this by setting the **a** element's **position** to **relative** and its **top** property to 0px, then changing the value of the **top** property for the active state. This moves the link 3 pixels away from the top edge (in other words, down):

```
a {
  ...
  position: relative;
  top: 0px;
}
a:active {
  top: 3px;
}
```

3. Logically, if the button were pressed down, there would be less room for the shadow, so we'll reduce the **box-shadow** distance as well:

```
a:active
  top: 3px;
  box-shadow: 0 1px 2px rgba(0,0,0,.5);
}
```

4. Save the file and give it a try in your browser. The links should turn green and appear to be pressed down when you click or tap them. That's not bad, but we can enhance the experience by adding some smooth transitions.

5. Make the background and border color transition ease in over .2 seconds, and see how that changes the experience of using the menu. We're using the shorthand **transition** property to keep the code simple. We're also using the default **ease** timing function at first, so we can omit that value:

```
a {
  ...
  transition: background-color .2s,
              border-color .2s;
}
```

6. Save your document, reload it in the browser, and try moving your mouse over the links. Do you agree it feels nicer? Now I'd like you to try some other duration values. See if you can still see the difference with a .1s duration. Now try a full second (1s). You'll find that 1 second is surprisingly slow. Try setting it to several seconds and trying out various **timing-function** values (just add them after the duration times). Can you tell the difference? Do you have a preference? When you are done experimenting, set the duration back to .2 seconds.

7. Now let's see what happens when we add a transition to the downward motion of the link when it is clicked or tapped. Transition both the **top** and **box-shadow** properties, because they should move in tandem. Let's start with a .2s duration like the others:

```
a {
  transition:
    background-color .2s,
    border-color .2s,
    top .2s,
    box-shadow .2s;
}
```

   Save the file, reload it in your browser, and try clicking the links. That transition really changes the experience of using the menu, doesn't it? The buttons feel more difficult to "press." Try increasing the duration. Does that make it feel even more difficult? It's interesting to see the effect that timing has on the experience of a user interface. It's important to get it right and not make things feel sluggish. A very short transition, such as .1s—or even no transition at all—would keep these buttons feeling snappy.

8. If you thought increasing the duration made the menu uncomfortable to use, try adding a short .5s delay to the **top** and **box-shadow** properties:

```
a {
  transition:
    background-color .2s,
    border-color .2s,
    top .1s .5s,
    box-shadow .1s .5s;
}
```

   I think you'll find that little bit of extra time makes the whole thing feel broken. Timing is everything!

CSS TRANSFORMS

CSS Transforms give authors a way to rotate, relocate, resize, and skew HTML elements in both two- and three-dimensional space (see **Note**). It is worth noting up front that transforms change how an element displays in a way that is neither motion- nor time-based; however, you can animate from one transform state to another using transitions or keyframe animations.

CSS Transforms Module Level 1 (2012) introduced an all-purpose **transform** property that is used with a number of transformation functions, including **scale()**, **translate()**, **rotate()**, and **skew()**. In 2020, the W3C drafted CSS Transforms Module Level 2, which introduced new and independent transformation properties (**scale**, **translate**, and **rotate**) aimed at simplifying the syntax and making animations less prone to problems.

That leaves us today with two acceptable approaches to creating basic 2D transformations. The original, all-purpose **transform** property is supported on virtually all browsers, so it's the safest bet. It is also still required for certain transformation types, like skew. The independent properties are well supported by browser versions released after mid-2022, but earlier browser releases won't recognize them. This is one of those slightly awkward transition periods we go through in the web development community from time to time as we move from one standard to another. As of this writing, most developers feel comfortable using the new properties along with a fallback method that uses the **transform** property. That may change by the time you are reading this, so as usual, check ***CanIUse.com*** for updated support information.

NOTE

I'll be focusing on 2D transforms in this chapter, as you are unlikely to need to delve into the more complicated 3D transforms as a beginner.

The transform Property

Let's start our discussion with the tried-and-true **transform** property. In the following sections, I'll introduce the independent property alternatives for each transformation type.

`transform`

Values:	`rotate()` | `translate()` | `translateX()` | `translateY()` | `scale()` | `scaleX()` | `scaleY()` | `skew()` | `skewX()` | `skewY()` | `none`
Default:	none
Applies to:	Transformable elements (see sidebar)
Inherits:	No

As mentioned earlier, you can use the **transform** property to rotate, relocate (translate), scale, and skew an element. The value of **transform** is one of the functions listed above that pass it specific values. FIGURE 20-6 shows a representation of four two-dimensional transform functions: **rotate()**, **translate()**, **scale()**, and **skew()** (see **Note**). The dashed outline shows the element's original size and position.

NOTE

There are actually five 2D transform functions in the Level 1 module. The fifth, `matrix()`*, allows you to craft your own combined transformation using six values and some badass trigonometry.*

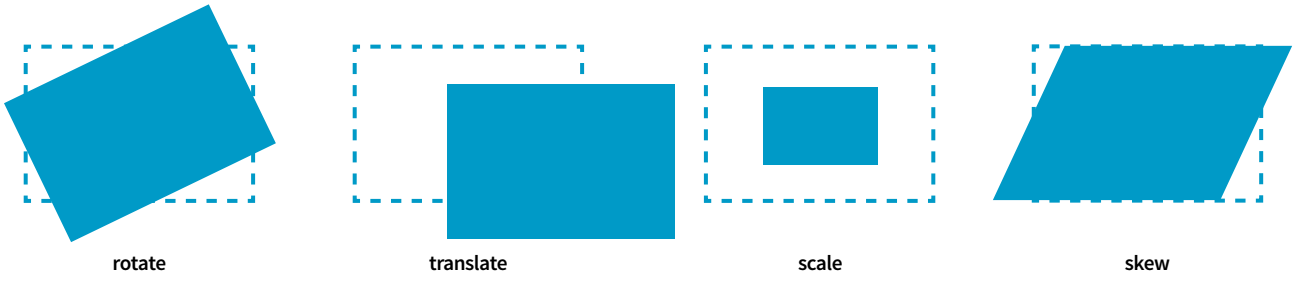


FIGURE 20-6. Four types of transforms: `rotate()`, `translate()`, `scale()`, and `skew()`

Transformable Elements

You can apply the `transform` property to most element types, including:

- HTML elements with replaced content, such as `img`, `canvas`, form inputs, and embedded media
- Elements with their display set to `block`, `inline-block`, `inline-table` (or any of the `table-*` display types), `grid`, and `flex`

It may be easier to note the element types you *cannot* transform, which include:

- Non-replaced inline elements, like `em` or `span`
- Table columns and column groups (but who'd want to?)

When you apply a transform function to an element in its normal state, it appears in its transformed state when the page loads. More commonly, a transformation is triggered when users interact with the element, via `:hover` or a JavaScript event. For example, you can make a button grow slightly larger when the pointer is over it.

When an element transforms, its element box keeps its original position and influences the layout around it, in the same way that space is left behind by a relatively positioned element. It is as though the transformation magically picks up the pixels of the rendered element, messes around with them, and lays them back down on top of the page. So, if you move an element with `transform`, you're moving only a picture of it. That picture has no effect on the surrounding layout.

Let's go through the transformation types one by one, starting with rotation.

Transforming the Angle (rotate)

If you'd like an element to appear on a bit of an angle, use `transform` with the `rotate()` function or the independent `rotate` property.

We'll start with the `transform` property option. The value of the `rotate()` function is an angle, which can be specified in positive (clockwise) or negative (counterclockwise) degrees (see **Note**).

The image in FIGURE 20-7 has been rotated –10 degrees (350 degrees) with the following style rule. The tinted image shows the element's original position for reference:

```
img {
  width: 400px;
  height: 300px;
  transform: rotate(-10deg);
}
```

You can achieve the same effect using the independent `rotate` property introduced in CSS Transforms Module Level 2.

NOTE

There are other units that may be used for specifying angles (`grad`, `rad`, and `turn`), but I will be sticking with the more common degree (`deg`) unit in this chapter. For more information, see w3.org/TR/css3-values/#angle-value.

```
transform: rotate(-10deg);
or
rotate: -10deg;
```

FIGURE 20-7. Rotating an `img` element by using `transform: rotate()` or the `rotate` property

rotate

Values: *angle* | none (there are additional values appropriate for 3D transformations; see the spec for more information)

Default: none

Applies to: Transformable elements

Inherits: No

This example uses the **rotate** property to reproduce the effect shown in FIGURE 20-7. The unit (**deg** in this case) is required for **rotate**:

```
img {
  width: 400px;
  height: 300px;
  rotate: -10deg;
}
```

Notice that the image rotates around its center point, which is the default point around which all transformations happen. But you can change that easily with the **transform-origin** property (FIGURE 20-8).

transform-origin

Values: *percentage* | *length* | `left` | `center` | `right` | `top` | `bottom`

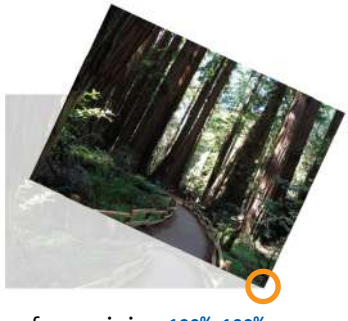
Default: 50% 50%

Applies to: Transformable elements

Inherits: No

The value for **transform-origin** is a set of two keywords, length measurements, or percentage values. The first value is the horizontal offset, and the second is the vertical offset. If only one value is provided, it will be used for both. If I wanted to rotate our image around a point at the center of its top edge, I could write it in any of the following ways:

```
transform-origin: center top;
```



```
transform-origin: 100% 100%;
```

```
transform-origin: 400px 0;
```

FIGURE 20-8. Changing the point around which the image rotates by using `transform-origin`

```
transform-origin: center top;

transform-origin: 100%, 100%;

transform-origin: 400px, 0;
```

The images in FIGURE 20-8 have all been rotated 25 degrees, but from different origin points. It is easy to demonstrate the origin point with the **rotate()** function, but keep in mind that you can set an origin point for any of the transform functions. **transform-origin** works with the independent transform properties, like **rotate**, as well.

Transforming the Position (translate)

Another way to transform an element is to give it a new location on the page by using one of the three translate functions (**translateX()**, **translateY()**, or **translate()**) as a value for **transform**, or by using the newer **translate** property. FIGURE 20-9 shows the effects of using **transform: translate()** or **translate** to move an element horizontally, vertically, or both.

```
transform: translate(90px, 60px);
or
translate: 90px 60px;
```

```
transform: translate(-5%, -25%);
or
translate: -5% -25%;
```

FIGURE 20-9. Moving an element around with the **translate()** function or the **translate** property

Using the **transform** property, the **translateX()** function allows you to move an element on a horizontal axis. **translateY()** is for moving along the vertical axis, and **translate()** combines both *x* and *y* values, separated by commas:

```
transform: translateX(50px);

transform: translateY(25px);

transform: translate(50px, 25px); /* (translateX, translateY) */
```

You can provide length values in any of the CSS units, or as a percentage. Percentages are calculated based on the width of the **border box** by default—that is, from border edge to border edge (see the sidebar **"The Transform Reference Box"**). You can provide positive or negative values, as shown in FIGURE 20-9.

If you provide only one value for the shorthand **translate()** function, it will be presumed to be the **translateX** value, and **translateY** will be set to 0. So, **translate(20px)** would be equivalent to applying both **translateX(20px)** and **translateY(0)**.

The newer independent property, **translate**, can be used as an alternative to the **translate()** function.

translate

Values:	none | *[x-axis length or percentage] [y-axis length or percentage] [z-axis length]*
Default:	none
Applies to:	Transformable elements
Inherits:	No

The values for the **translate** property work a little differently from the values for the function. If you provide one value, it applies to the x (horizontal) axis. This example moves an element 50 pixels to the right, as I did in the previous example:

```
translate: 50px; /* (x-axis) */
```

Values for the x-axis may also be provided as percentages. When you provide two values (separated by character spaces), the first is for the x-axis, and the second is for the y (vertical) axis. These examples replicate the transformation examples shown earlier (percentage values may also be used for either value):

```
translate: 0px 25px; /* (x-axis y-axis) */
translate: 50px 25px; /* (x-axis y-axis) */
```

When a third value is provided, it applies to the z-axis, allowing the **translate** property to position elements in three-dimensional space. The value for the z-axis may only be a length, not a percentage:

```
translate: 50px 25px 50px; /* (x-axis y-axis z-axis) */
```

CSS Transforms Module Level 2 provides a comprehensive toolbox for 3D transforms, which, unfortunately, is beyond the scope of this chapter.

Transforming the Size (scale)

You can make an element appear larger or smaller by using one of three scale functions with **transform**—**scaleX()** (horizontal), **scaleY()** (vertical), and the shorthand **scale()**—or by using the independent **scale** property.

The value for the **scaleX()** and **scaleY()** functions is a unitless number that specifies a size ratio. This example makes an image 150% of (1.5 times) its original width:

```
a img {
  transform: scaleX(1.5);
}
```

The Transform Reference Box

By default, CSS Transforms affect the border box of the transformed HTML element—that is, the element box including any border that may be applied, but excluding the element's margin. You can choose an alternate reference box to transform with the **transform-box** property. Its values are **content-box**, **border-box**, **fill-box**, **stroke-box**, and **view-box** (relevant to SVG images). Chances are that the default behavior will work fine for most uses, but it is good to know that there are other settings.

The **scale()** shorthand lists a value for **scaleX()** and a value for **scaleY()**. This example makes an element twice as wide but half as tall as the original:

```
a img {
  transform: scale(2, .5); /* scaleX, scaleY */
}
```

Unlike with **translate()**, if you provide only one value for **scale()**, it will be used as the scaling factor in both directions. So, specifying **scale(2)** is the same as applying **scaleX(2)** and **scaleY(2)**, which is intuitively the way you'd want it to be.

FIGURE 20-10 shows a few examples of **transform** used with the **scale()** function. It also shows how you can achieve the same effects with the newer **scale** property.

Let's take a look at the syntax for that property now.

scale

Values:	none | *1 to 3 length or percentage values (applied to x-, y-, and z-axes)*
Default:	none
Applies to:	Transformable elements
Inherits:	No

As for **translate**, if you provide one value, it applies to the x (horizontal) axis. Two values apply to the x- and y-axes, in that order. A third value applies to the z-axis for 3D transformations. All three values for **scale** may be a unitless scaling number (as for the **scale()** function), or a percentage. Values are separated by a character space. Here are some of the ways you could achieve the same effects as in the **transform: scale()** examples shown earlier using the **scale** property instead:

```
transform: scale(1.25);
or
scale: 1.25;
or
scale: 125%;
```

```
transform: scale(.75);
or
scale: .75;
or
scale: 75%;
```

```
transform: scale(1.5, .5);
or
scale: 1.5 .5;
or
scale: 150% 50%;
```

FIGURE 20-10. Changing the size of an element with the **scale()** function or the **scale** property

```
a img {
  scale: 1.5;
}

a img {
  scale: 2 50%;
}
```

Making It Slanty (skew)

The quirky collection of skew functions for the **transform** property—**skewX()**, **skewY()**, and the shorthand **skew()**—change the angle of either the horizontal or the vertical axis (or both) by a specified number of degrees. As for **translate()**, if you provide **skew()** only one value, it is used for **skewX()**, and **skewY()** will be set to zero. There is no independent property for skewing elements at the moment, but it may be added in the future.

The best way to get an idea of how skewing works is to take a look at some examples (FIGURE 20-11):

```
a img {
  transform: skewX(15deg);
}

a img {
  transform: skewY(30deg);
}

a img {
  transform: skew(15deg, 30deg);
}
```

transform: skewX(15deg);

transform: skewY(30deg);

transform: skew(15deg, 30deg);

FIGURE 20-11. Slanting an element by using the skew() function

Applying Multiple Transforms

It is possible to apply more than one transformation function to a single element by listing out the functions and their values, separated by spaces, like this:

```
transform: function(value) function(value);
```

In this example, I've made the forest image get larger, tilt a little, and move down and to the right when the mouse is over it or when it is in focus. FIGURE 20-12 shows the result:

```
img:hover, img:focus {
  transform: scale(1.5) rotate(-5deg) translate(50px,30px);
}
```

Know that transforms are applied in the order in which they are listed. For example, if you apply **translate()** and then **rotate()**, you get a different result than if you apply **rotate()** and then **translate()**. Order matters.

Another thing to watch out for is that if you want to apply an additional transform on a different state (such as **:hover**, **:focus**, or **:active**), you need to repeat all of the transforms already applied to the element. For example,

FIGURE 20-12. Applying `scale()`, `rotate()`, and `translate()` to a single element

this `a` element is rotated 45 degrees in its normal state. If I applied a `scale()` transform on the `:hover` state, like this, I would lose the rotation:

```
a {
  transform: rotate(45deg);
}

a:hover {
  transform: scale(1.25); /* the rotation on a element will be lost */
}
```

To achieve both rotation and scale, I need to provide both transform values:

```
a:hover {
  transform: rotate(45deg) scale(1.25);  /* rotates and scales */
}
```

This should give you a glimpse of how the need for redundancy and the risk of accidentally overriding earlier transformation values make the `transform` property a little cumbersome to use, especially for complex animations with many changing parts. When the transformations are controlled by separate properties, as in the following example, they are easier to change without affecting other settings. This was exactly the motivation for rolling out the new properties in Level 2:

```
a:hover {
  rotate: 45deg;
  scale: 1.25;
}
```

Smoooooth Transforms

The multiple transforms applied to the redwood forest image look interesting, but it might *feel* better if we got there with a smooth animation instead of just BAM! Now that you know about transitions and transforms, let's put them together and make some magic happen. By "magic," of course, I mean some basic animation effects between two states. We'll do that together, step by step, in EXERCISE 20-2.

EXERCISE 20-2. Transitioning transforms

In this exercise, we'll make the food photos in the gallery shown in FIGURE 20-13 grow and spin out to an angle when the user mouses over them—and we'll make it smoooooth with a transition. A starter document (*transforms.html*) and all of the images are available with the materials for this chapter. We'll be using the better-supported **transform** property in this exercise, but feel free to try out the independent properties as you go along.

1. Open *transforms.html* in a text or code editor, and you will see that there are already styles that arrange the list items horizontally and apply a slight drop shadow. The first thing we'll do is add the **transform** property for each image.

2. We want the transforms to take effect only when the mouse is over the image or when the image has focus, so the **transform** property should be applied to the **:hover** and **:focus** states. Because we want each image to tilt a little differently, we'll need to write a rule for each one, using its unique ID as the selector. You can save and check your work when you're done.

   Note: You could use the independent **rotate** property if you prefer and if your browser is up to date.

   ```
   a:hover #img1, a:focus #img1 {
     transform: rotate(-3deg);
   }
   a:hover #img2, a:focus #img2 {
     transform: rotate(5deg);
   }
   a:hover #img3, a:focus #img3 {
     transform: rotate(-7deg);
   }
   a:hover #img4, a:focus #img4 {
     transform: rotate(2deg);
   }
   ```

3. Now let's make the images a little larger as well, to give visitors a better view. Add **scale(1.5)** to each of the **transform** values (or add a **scale** property declaration). Here is the first one; you do the rest:

   ```
   a:hover #img1, a:focus #img1 {
     transform: rotate(-3deg) scale(1.5);
   }
   ```

   Note that the image files are created at the larger size and then scaled down for the thumbnail view. If I had started with small images and scaled them larger, they would look fuzzy.

4. As long as we are giving the appearance of lifting the photos off the screen, let's make the drop shadow appear to be a little farther away by increasing the offset and blur, and lightening the shade of gray. All images should have the same effect, so add one rule using **a:hover img** as the selector:

   ```
   a:hover img, a:focus img {
     box-shadow: 6px 6px 6px rgba(0,0,0,.3);
   }
   ```

   Save your file and check it out in a browser. The images should tilt and look larger when you mouse over them. But the action is kind of jarring. Let's fix that with a transition.

5. Add the **transition** shorthand property to the normal **img** state (i.e., not on **:hover** or **:focus**). The property we want to transition in this case is **transform**. Set the duration to .3 seconds and use the **linear** timing function:

   ```
   img {
     ...
     transition: transform .3s linear;
   }
   ```

   Note: If you have been using independent properties, you will need to repeat the transition for each of them. See the example earlier in the chapter for transitioning several properties at once.

And that's all there is to it! You can try playing around with different durations and timing functions, or try altering the transforms or their origin points to see what other effects you can come up with.

FIGURE 20-13. Photos get larger and tilt on **:hover** and **:focus**. A transition is used to help smooth out the changes between states.

3D Transforms

In addition to the two-dimensional transform functions we've just seen, the CSS Transforms spec also describes a system for creating a sense of three-dimensional space and perspective. Combined with transitions, you can use 3D transforms to create rich interactive interfaces such as image carousels, flippable cards, or spinning cubes! FIGURE 20-14 shows a few examples of interfaces created with 3D transforms.

Animated book covers by Marco Barria
tympanus.net/Development/AnimatedBooks/

Webflow transform tools example
3d-transforms.webflow.com

Movie poster animation by Marco Kuiper
demo.marcofolio.net/3d_animation_css3/

Isometric Color Picker by Jon Kantner
codepen.io/jkantner/pen/xxXmVKw

FIGURE 20-14. Some examples of 3D transforms. The book covers, movie posters, and 3D cube also have cool animation effects, so it's worth going to the links and checking them out.

It's worth noting that this method does not create 3D objects with a sense of volume; it merely tilts the otherwise flat element boxes around on three axes (animation expert Val Head calls them "postcards in space"). That said, 3D transforms add some interesting depth to an otherwise flat web page.

3D transforms are not a need-to-know skill for folks just starting out in web design, but if you'd like to learn more, the following tutorials are good places to start. Although they were both published more than 10 years ago, they remain some of the best how-to articles on the topic. Just ignore the out-of-date browser support notes:

- "Adventures in the Third Dimension: CSS 3D Transforms" by Peter Gasston (*oreil.ly/ZkV00*)
- "Intro to CSS 3D Transforms" by David DeSandro (*3dtransforms.desandro.com*)

If you'd like to see some nice, subtle examples with code you can copy and use yourself, check out the collection at *polypane.app/css-3d-transform-examples/*.

The most detailed reference on using the new independent properties for 3D transforms is CSS Transforms Module Level 2 itself (*w3.org/TR/css-transforms-2*); however, be forewarned that it's not written with beginners in mind.

KEYFRAME ANIMATIONS

The CSS Animations Level 1 spec allows authors to create real, honest-to-goodness keyframe animations. FIGURE 20-15 shows just a few examples that you can see in action online. Unlike transitions that go straight from a beginning state to an end state, keyframe animations allow you to explicitly specify other states at points along the way, allowing for more granular control of the action. Those "points along the way" are established by **keyframes** that define the beginning or end of a segment of animation.

Creating keyframe animations is too complex for me to cover in depth here, but I'd like you to have a general idea of how it works, so I'll sketch out the minimal details. See the **"CSS Animation Resources"** sidebar if you'd like to continue your keyframe animation journey.

The animation process has two parts:

1. Establishing the keyframes (the points at which something in the animation changes) with an `@keyframes` rule
2. Adding the animation properties to the elements that will be animated

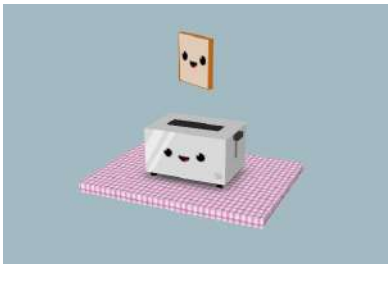
> **TERMINOLOGY**
>
> Keyframe animation is known as **explicit animation** because you program its behavior. By contrast, transitions are an example of **implicit animation** because they are triggered only when a property changes.

Establishing the Keyframes

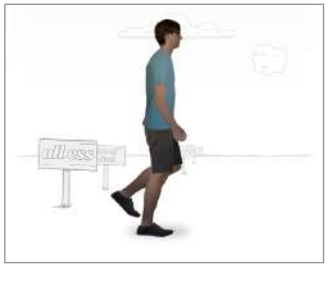
The `@keyframes` at-rule identifies the name of the animation, the stages of the animation (represented by percentage values), and the CSS properties that are affected for each stage. Here's what an `@keyframes` rule looks like abstracted down to its syntax:

```
@keyframes animation-name {
  keyframe { property: value; }
  /* additional keyframes */
}
```

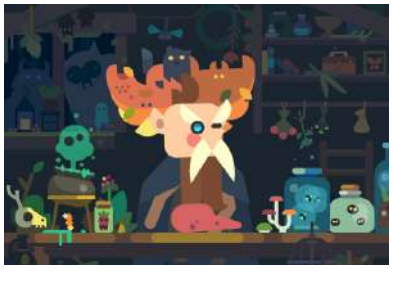
Here is a very simple set of keyframes that change the background color of an element over time. It's not a very action-packed animation, but it should give you a basic understanding of what an `@keyframes` rule does:



CSS Animated 3D toaster
by JHEY
codepen.io/jh3y/pen/KKVjLrx



How I Learned to Walk
by Andrew Wang-Hoyer
andrew.wang-hoyer.com/experiments/walking/



Pure CSS "Moustached Nanny"
by Julia Miocene
codepen.io/miocene/pen/mjLPVp

FIGURE 20-15. Examples of animations using only CSS

FIGURE 20-16. Animating through the colors of the rainbow by using keyframes

```
@keyframes colors {
  0% { background-color: red; }
  20% { background-color: orange; }
  40% { background-color: yellow; }
  60% { background-color: green; }
  80% { background-color: blue; }
  100% { background-color: purple; }
}
```

This sample **`@keyframes`** rule says: create an animation sequence called "colors". At the beginning of the animation, the **`background-color`** of the element should be red; at 20% through the animation runtime, the background color should be orange; and so on, until it reaches the end of the animation. The browser fills in all the shades of color in between each keyframe (or *tweens* it, to use the lingo). This is represented the best I could in FIGURE 20-16.

Each percentage value and corresponding property/value declaration defines a keyframe in the animation sequence.

As an alternative to percentages, you can use the keyword **`from`** for the start of an animation sequence (equivalent to 0%) and the keyword **`to`** for denoting the end (100%). The following example makes an element slide in from right to left as the left margin reduces to 0:

```
@keyframe slide {
  from { margin-left: 100% }
  to { margin-left: 0%; }
}
```

CSS Animation Resources

The following resources are good starting points for learning more about CSS keyframe animation:

- *Designing Interface Animation: Meaningful Motion for User Experience* by Val Head (Rosenfeld Media).
- *Transitions and Animations in CSS* by Estelle Weyl (O'Reilly).
- *Creating Web Animations: Bringing Your UIs to Life* by Kirupa Chinnathambi (O'Reilly).
- The CSS Animations Level 2 spec (*w3.org/TR/css-animations-2/*).
- "Animation & UX Resources" by Val Head (*valhead.com/ui-animation/*). Val has compiled a mega-list of resources about web animation, including links to tutorials, articles, tools, galleries, and more. It is not limited to CSS keyframe animation, but as long as you're delving into animation, you can trust my friend Val to point you to good stuff.
- "CSS Animation for Beginners" by Rachel Cope (*oreil.ly/9i_ap*). This is a clearly written tutorial with lots of examples.
- "The Guide to CSS Animation: Principles and Examples" by Tom Waterhouse (*oreil.ly/k8qHE*). This tutorial goes beyond CSS code to include tips for creating natural animation effects.

Adding Animation Properties

Now we can apply this animation sequence to an element or multiple elements in the document by using a collection of animation properties that are very similar to the set of transition properties that you already know.

We're going to apply the "colors" animation to the `#magic div`:

```
<div id="magic">Magic!</div>
```

In the CSS rule for `#magic`, we make the following decisions about the animation we want to apply:

- Which animation to use (`animation-name`) (*required*).
- How long it should take (`animation-duration`) (*required*).
- The manner in which it should accelerate (`animation-timing-function`). This property uses the same timing function keywords that we covered for CSS Transitions.
- Whether to pause before it starts (`animation-delay`).

Looks familiar, right? There are a few other animation-specific properties to know about as well:

`animation-iteration-count`

: How many times the animation should repeat. This can be set to a whole number or `infinite`.

`animation-direction`

: Whether the animation plays forward (`normal`), in reverse (`reverse`), alternates back and forth starting at the beginning (`alternate`), or alternates starting from the end (`alternate-reverse`).

`animation-fill-mode`

: What happens with the animation before it begins and after it ends. By default (`none`), the animation shows whatever property values were not specified via `@keyframes`. If you want the last keyframe to stay visible after the animation plays, use the `forwards` keyword. If there is a delay set on the animation and you want the first keyframe to show during that delay, use `backwards`. To retain the beginning and end states, use `both`.

`animation-play-state`

: Whether the animation should be `running` or `paused` when it loads. The play state can be toggled on and off based on user input with JavaScript or on hover.

The required `animation-name` property tells the browser which keyframe sequence to apply to the `#magic div`. I've also set the duration (required) and timing function, and used `animation-iteration-count` to make it repeat infinitely. I could have provided a specific number value, like 2 to play it twice, but how fun are only two rainbows? And for fun, I've set the `animation-direction`

to **alternate**, which makes the animation play in reverse after it has played forward. Here is the resulting rule for the animated **div**:

```
#magic {
  ...
  animation-name: colors;
  animation-duration: 10s;
  animation-timing-function: linear;
  animation-iteration-count: infinite;
  animation-direction: alternate;
}
```

That gets a bit verbose, so you can also use the **animation** shorthand property to combine the values, just as we did for **transition**:

```
#magic {
  animation: colors 10s linear infinite alternate;
}
```

Those are the bare bones of creating keyframes and applying animations to an element on the page. To make elements move around (what we typically think of as "animation"), use keyframes to change the position of the element on the screen with **translate** (the best option for performance) or with the **top**, **right**, **bottom**, and **left** properties. When the keyframes are tweened, the object will move smoothly from position to position. You can also animate the other transform functions, such as **scale()** and **skew()** or their independent property equivalents.

When to Use Keyframe Animation

To keep the example simple, I chose to change only the background color of a button element—but of course, keyframe animations can be used to create

Animation Inspectors

All the modern browsers offer tools to inspect and modify web animations (FIGURE 20-17). When you inspect an animated element in your browser's developer tools, click the Animations tab to see a timeline of all the animations applied to that element. You can slow down the animation to reveal what is happening on a detailed level. You can also modify the animation by making changes to the timing, delay, duration, and keyframes. For more information on accessing and using the animation inspector in each browser, see the following:

- Chrome (*oreil.ly/nwywz*)
- Safari (*oreil.ly/KTwhf*)
- Firefox (*oreil.ly/nZSP5*)
- Microsoft Edge (*oreil.ly/pCk0b*)

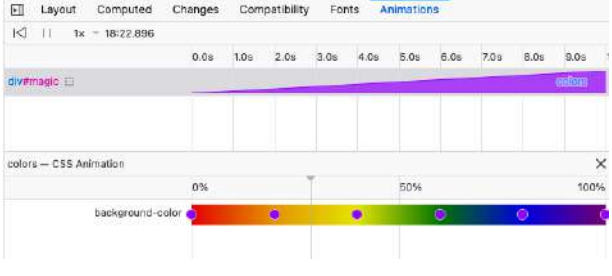


FIGURE 20-17. The "colors" animation example shown in Firefox's animation inspector

more impressive animations, especially when combined with the CSS transform functions for spinning and moving elements around on the page. If you only need to change an element from one state to another, a transition is the way to go. But if you have a linear animation with more than one state, such as moving a character, an object, or its parts around, keyframe animation is the most appropriate choice.

For more complex keyframe animations, particularly those that change with user interaction or require complex physics, using JavaScript for animation may be a better choice than CSS animation. CSS keyframe animation is a good solution for simple animations used as an enhancement to a baseline experience. If you'd like some shortcuts for basic UI animation effects, you can try downloading one of the premade animations from the sites listed in the **"Animation Tools"** sidebar.

Need a Little Inspiration?

It's both fun and educational to see what folks out there are doing with CSS animation. Here are a couple of sites that will give you plenty of ideas:

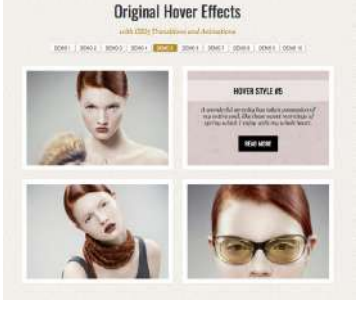
CodePen (codepen.io)

CodePen is an online tool for writing code, but it is also a community where people share what they are working on. A search for "animation" will turn up lots of cool examples, with the added benefit of showing you the code that drives them. It's a great way to learn.

Codrops (tympanus.net/codrops/)

The Codrops blog is a treasure trove of examples of CSS transitions, transforms, and animations. Check out the Demos section for a gallery of cool experiments. The Original Hover Effects tutorial (FIGURE 20-18) includes the code for a variety of transition and transform tricks and is available at *oreil.ly/PT1ff*.

Codepen.io



Codrops blog (tympanus.net/codrops/)

FIGURE 20-18. A sample animation "pen" on Codepen.io (left) and a hover tutorial on the Codrops blog (right)

Animation Tools

If you want to add a simple animation effect to an element—a quick flip here or a little shimmy there—you may be able to find a premade effect you can apply to your design. Here are a few sites that provide ready-made CSS for common animation effects (some also use JQuery [JavaScript] plug-ins, but they explain how to use them):

- **Animate.css** by Daniel Eden (*animate.style*)
- **CSS Animation Cheat Sheet** by Justin Aguilar (*justinaguilar.com/animations/*)
- **AngryTools CSS Animation Kit** (*angrytools.com/css/animation/*)

WRAPPING UP

I hope I've helped you to wrap your head around how CSS can be used to add a little motion and smoothness to your pages. For adding motion to a web page, we have CSS Transitions to smooth out changes from one state to another and CSS Animations for animating a series of states. Used thoughtfully, animation can make your interfaces more intuitive and enhance your brand personality. We also looked at CSS Transforms for repositioning, rotating, resizing, or skewing an element when it is rendered on the screen.

Now let's see if you were paying attention with a little quiz!

TEST YOURSELF

Think you know your way around transitions, transforms, and keyframe animations? Here are a few questions to find out. (Answers are provided in **Appendix A.**)

1. What is tweening?

2. If a transition had keyframes, how many would it have?

3. Write out the transition declaration (property and value) you would use to accomplish each of the following:
   a. Wait .5 seconds before the transition starts.
   b. Make the transition happen at a constant speed.
   c. Make the transition last .5 seconds.
   d. Make the lines of text slowly grow farther apart.

4. Which of the following can you *not* animate?
   a. `width`
   b. `padding`
   c. `text-transform`
   d. `word-spacing`

5. Which timing function will be used if you omit the **`transition-timing-function`** property? Describe its action.

6. In the following transition, what does .2s describe?

```
transition: color .2s linear;
```

7. Which transition will finish first?

   a. `transition: width 300ms ease-in;`

   b. `transition: width 300ms ease-out;`

8. Write the **`transform`** declarations to accomplish the following:

   a. Tilt the element 7 degrees clockwise.

   b. Reposition the element 25 pixels up and 50 pixels to the left.

   c. Rotate the element from its bottom-right corner.

   d. Make a 400-pixel-wide image display at 500 pixels wide.

9. In the following transform declaration, what does the 3 value describe?

```
transform: scale(2, 3)
```

10. What happens halfway through this animation?

```
@keyframes border-bulge {
  from { border-width: 1px; }
  25% { border-width: 10px; }
  50% { border-width: 3px; }
  to { border-width: 5px; }
}
```

11. Write the animation declarations you would use to accomplish the following:

    a. Make the animation play in reverse.

    b. Make the entire animation last 5 seconds.

    c. Wait 2 seconds before running the animation.

    d. Repeat the animation three times and then stop.

CSS REVIEW: TRANSITIONS, TRANSFORMS, AND ANIMATION

Here is a summary of the properties covered in this chapter.

Property	Description
`animation`	Shorthand property that combines animation properties
`animation-delay`	Indicates the amount of time before the animation starts running
`animation-direction`	Specifies whether the animation plays forward, in reverse, or alternates back and forth
`animation-duration`	Specifies the amount of time the animation lasts
`animation-fill-mode`	Overrides limits to when animation properties can be applied
`animation-iteration-count`	Indicates the number of times the animation repeats
`animation-name`	Specifies the named animation sequence to apply
`animation-play-state`	Specifies whether the animation is running or paused
`animation-timing-function`	Describes the acceleration of the animation
`rotate`	Rotates the element
`scale`	Changes the dimensions of the element
`transform`	Specifies that the rendering of an element should be altered via one of the 2D or 3D transform functions
`transform-origin`	Denotes the point around which an element is transformed
`transform-style`	Preserves a 3D context when transformed elements are nested
`transition`	Shorthand property that combines transition properties
`transition-delay`	Specifies the amount of time before the transition starts
`transition-duration`	Specifies the amount of time the transition animation lasts
`transition-property`	Defines which CSS property will be transitioned
`transition-timing-function`	Describes the manner in which the transition happens (changes in acceleration rates)
`translate`	Repositions the element horizontally and/or vertically

CHAPTER
21

MORE CSS TECHNIQUES

IN THIS CHAPTER

Styling tips for forms

Using a CSS reset or normalizer

Strategies for supporting older browsers (@supports)

Creating your own custom properties

Nesting CSS rules

Setting priority with CSS layers

CSS frameworks

By now you have a solid foundation in writing style sheets. You can style text and element boxes, float and position objects, create responsive page layouts using Flexbox and Grid Layout, and even add subtle animation effects to your designs. But there are a few more CSS techniques and features that you should know about.

If you look at the **"In This Chapter"** list, you'll see that this chapter is a grab bag of sorts. It starts with general approaches to styling forms. Then you'll learn how to clear out default browser styles with a CSS reset. We'll also discuss strategies for supporting older browsers, including using `@supports` to test for the support of a particular feature. Finally, we'll round out the chapter with introductions to some newer CSS features like variables, nesting, and layers as well as a quick introduction to CSS frameworks. Let's dig in!

STYLING TIPS FOR FORMS

Web forms can look a bit hodgepodge right out of the box with no styles applied (FIGURE 21-1), so you'll certainly want to give them a more professional appearance. Not only do they look better, but studies show that forms are easier and faster to use when the labels and inputs are lined up nicely. In this section, we'll look at how various form elements can be styled.

I'm not going to lie...styling forms is somewhat of a dark art because of the variety of ways in which browsers handle form elements. For really slick custom forms, you will generally need to turn to JavaScript. But the efforts are well worth it to improve usability.

For the most part, you can use the standard color, background, font, border, margin, and padding properties that you've learned in the previous chapters

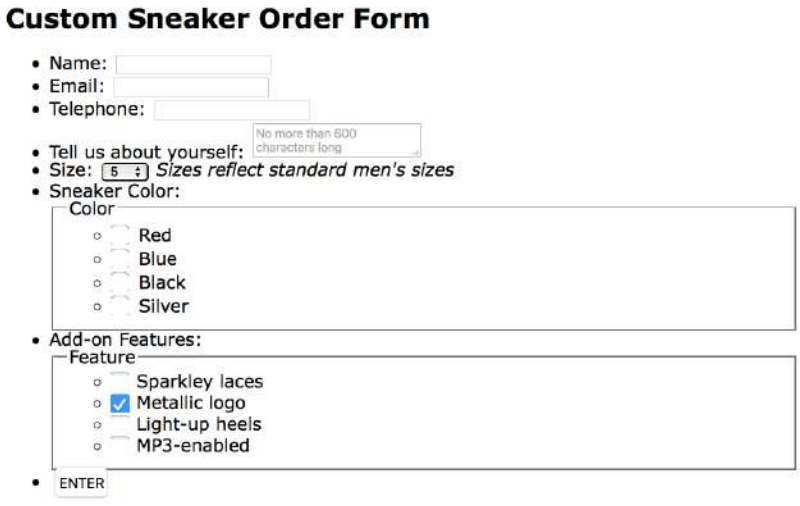


FIGURE 21-1. Forms created with HTML alone tend to be ugly and difficult to use. Don't worry—this one gets spiffed up in FIGURE 21-2.

to style aspects of form elements using CSS. The following is a quick rundown of the types of things you can do for each form control type:

Text inputs (`text`, `password`, `email`, `search`, `tel`, `url`)

Change the appearance of the box itself with **`width`**, **`height`**, **`background-color`**, **`background-image`**, **`border`**, **`border-radius`**, **`margin`**, **`padding`**, and **`box-shadow`**. You can also style the text inside the entry field with the **`color`** property and the various font properties.

The `textarea` *element*

`textarea` elements can be styled in the same way as text entry fields. They use a monospace font by default, so you may want to change that to match your other text entry fields. Because there are multiple lines in a **`textarea`**, you may also want to specify the line height. Note that some browsers display a handle on the lower-right corner of the **`textarea`** box that makes it resizable, but you can turn this behavior off by adding the style **`resize: none`**. Text areas display as **`inline-block`** by default, but you can change them to **`block`** with the **`display`** property.

Button inputs (`submit`, `reset`, `button`)

You can apply any of the box properties to button inputs (**`width`**, **`height`**, **`border`**, **`background`**, **`margin`**, **`padding`**, and **`box-shadow`**). It is worth noting that buttons are set to the **`border-box`** sizing model by default. Most browsers also add a bit of padding by default, which you can override with your own padding value. You can also style the text that appears on the buttons.

Radio and checkbox buttons

You can use the newer `accent-color` property to change the highlight color browsers apply to radio buttons and checkboxes. You can also change the size of radio and checkbox buttons in modern browsers other than Safari (as of this writing). It is possible to create custom buttons entirely with JavaScript as well. If you would like to learn more about customizing these button types with CSS, check out these articles:

- "Zero Trickery Custom Radios and Checkboxes" by Chris Coyier (*oreil.ly/Z1qs2*)
- "Pure CSS Custom Styled Radio Buttons" by Stephanie Eckles (*oreil.ly/Yg3QK*)
- "Pure CSS Custom Checkbox Style" by Stephanie Eckles (*oreil.ly/Ijiui*)

Drop-down and select menus (`select`)

You can specify the width and height for a `select` element, but note that it uses the `border-box` sizing model by default. Some browsers allow you to apply `color`, `background-color`, and font properties to `option` elements, but many developers choose to leave them alone to be rendered by the browser and operating system. For an overview of customizations you can make to drop-down menus, see "Creating a Custom <select> Dropdown with CSS" by Ibadehin Mojeed (*oreil.ly/qlt9h*), but be aware that some techniques are inconsistently handled by all but the most recent browsers.

Fieldsets and legends

You can treat a `fieldset` like any other element box, adjusting the border, background, margin, and padding. Turning the border off entirely is one way to keep your form looking tidy while preserving semantics and accessibility. By default, `legend` elements are positioned above the top border of the `fieldset`, and, unfortunately, browsers make them difficult to change. Some developers use a `span` or `b` element within the `legend` and apply styles to the contained element for more predictable results. Some choose to hide it in a way that ensures it will still be read by screen readers (`legend {width: 1px; height: 1px; overflow: hidden;}`).

Now we know what we can do to style individual controls, but the grander goal is to make the form more organized and easier to use. I've applied some basic styles to the form shown in FIGURE 21-1 to improve the organization and overall appearance. FIGURE 21-2 shows the resulting form in wide and narrow viewports. There are color, font, border, and spacing changes, and the labels and input elements are nicely aligned as well. And not only that, the form is responsive! I've used Flexbox to make the labels stack on top of their respective inputs and fieldsets on narrow screens so there is no wasted space.

New Form Properties

There is work being done at the W3C to give developers more control over the rendering of form widgets. CSS Basic Use Interface Module Level 4 (*w3.org/TR/css-ui-4/*) introduced two new properties, `accent-color` and `appearance`, that put style decisions in our hands.

The `accent-color` property specifies a color to be used for form elements that the browser typically highlights. It currently works with radio buttons, checkboxes, range inputs, and progress bars. Most modern browsers (with the exception of Safari, as of this writing) will change the color of the checkmark in a checkbox automatically to make sure it has adequate contrast against the chosen accent color.

The `appearance` property gives authors control over widgets that are typically tied to the UI style of the computer's operating system. Setting `appearance: none` clears the native styles to allow custom styles to be applied. This property is supported by the most recent browsers but may have unpredictable results in browsers released before 2022, so test your designs thoroughly should you decide to use it.

NOTE

Inputs for range, color, date, and file pickers cannot be changed with CSS alone. It is possible to customize them using JavaScript, but that is beyond the scope of this chapter.

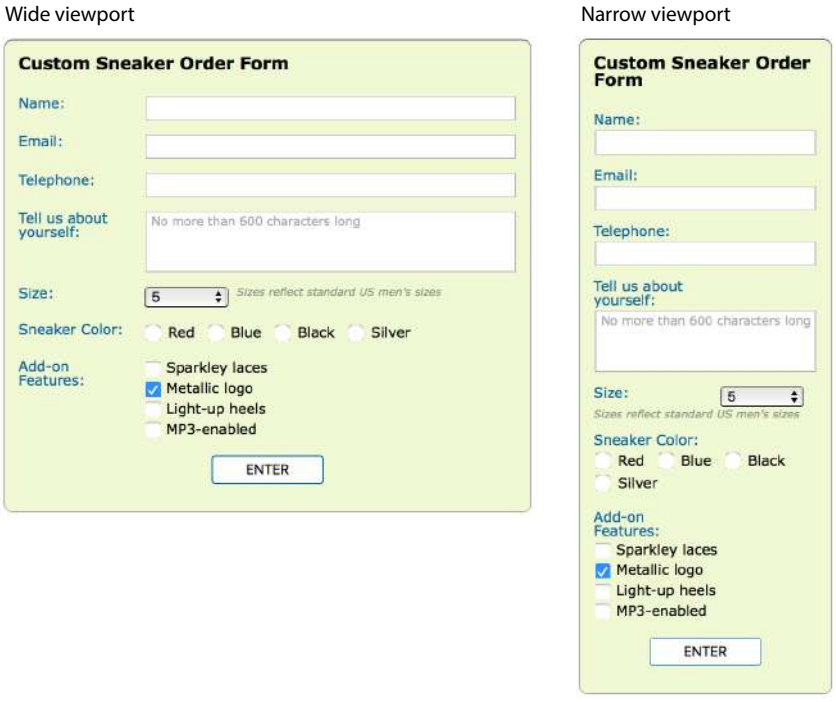


FIGURE 21-2. This responsive form uses Flexbox to allow text inputs to resize and to shift the position of the labels on small screens.

If you'd like to take a look at the actual markup and styles, the document *sneakerform.html* is available with the materials for this chapter (*learningwebdesign.com/6e/materials*). I've left thorough comments throughout that explain exactly what each style is for. The approach to styling the Custom Sneaker Order Form can be summarized as follows:

- Set the `box-sizing` property to `border-box` for the whole document. This makes sizing form elements more predictable.
- Give the `form` a `max-width` (so it can shrink to fit smaller viewports) and optional decorative styling, like the green background and rounded border in the example.
- Get rid of the bullets and spacing around the unordered lists that were used to mark up the form semantically.
- Turn list items (each containing a label and some sort of input or fieldset) into flex containers by setting their `display` property to `flex`. Turn on wrapping, which is what allows the inputs to shift below the labels on small screens.
- Give the labels fixed widths so they are sized the same regardless of screen size. Because labels on checkboxes and radio buttons work differently, set them to `flex: 1 1 auto`.

■ DESIGN TIP

Coordinated Fonts

By default, a browser may use different fonts at different sizes for various input types. For example, it may use the system font on buttons and a constant-width font like Courier for `textarea` inputs. If you'd like all inputs to use the same font as the surrounding text on the page, you can force the form elements to inherit font settings:

```
button, input, select,
textarea {
  font-family: inherit;
  font-size: 100%
}
```

- Allow the `input`, `textarea`, and `fieldset` elements to grow to fill the remaining space (`flex: 1 1 20em`). When the screen is too narrow for them to fit next to the labels, they wrap below.
- Set the text input fields' `font-family` property to `inherit` so they use the same font as the rest of the document instead of whatever font the browser uses for forms by default. Text inputs also get heights, borders, and a little padding.
- Fieldsets and legends are tricky to style. Turn off the border and padding on the `fieldset`, and then hide the `legend` in a way that ensures it will still be read aloud by screen readers before each checkbox or radio button option. Because there is both a label and a legend for each fieldset, I made sure they are not exactly the same so they won't be redundant when read aloud by a screen reader. The legend should be shorter because it is repeated for each option.
- Add a rounded border, background color, and font styling to the submit button. Set the side margins to `auto` so it will always be centered in the width of the form.

This is a very simple example, but it should give you a general idea of how forms can be styled. You may also want to add highlight styles for interactivity, such as `:hover` styles on the buttons and `:focus` styles for text inputs when they are selected.

Styling Tables

Like any other text content on a web page, content within table cells can be formatted with various font, text, and background properties. CSS Table Module Level 3 (*w3.org/TR/css-tables-3*) provides a number of table-specific styles for controlling the appearance and structure of the table itself, including `border-collapse`, `border-spacing`, `empty-cells`, `table-layout`, and `caption-side`. You can read about how to use them in my article "Styling Tables" (*learningwebdesign.com/articles/*).

A CLEAN SLATE (RESET AND NORMALIZE)

As you know, browsers have their own built-in style sheets (called **user agent style sheets**) for rendering HTML elements. If you don't supply styles for an `h1`, for example, you can be certain that it will display as large, bold text with space above and below—but just how large and how much space may vary from browser to browser. Furthermore, even if you do provide your own style sheet, elements in your document may be secretly inheriting certain styles from the user agent style sheets, causing unexpected results. If you encounter

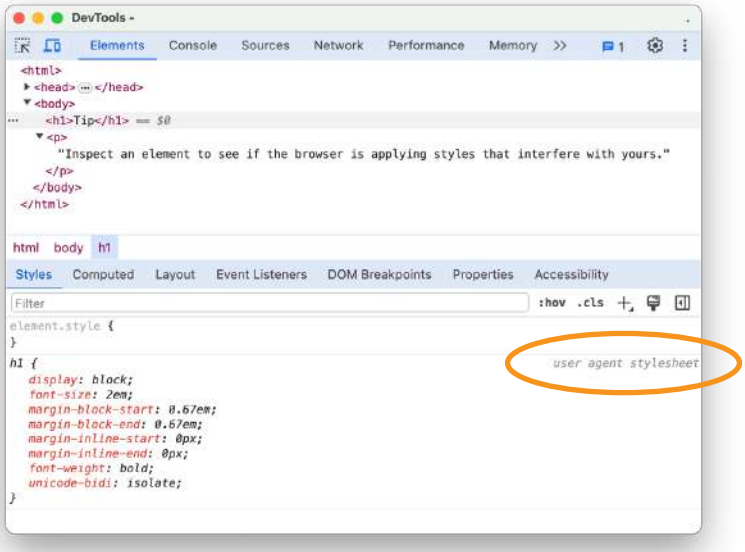


FIGURE 21-3. The element inspector exposes the styles coming in from the user agent style sheet. This example shows all of the styles applied to an `h1` by default in Chrome.

baffling style behavior, inspecting the element with your browser's developer tools to see all the applied styles may reveal the reason (FIGURE 21-3).

Many developers prefer to give their documents a neutral starting point for applying their own styles, to minimize browser inconsistencies. There are two methods for clearing out inherited styles: a CSS reset or a "normalizer." They take different approaches, so one or the other may be the best solution for what you need to achieve. Of course, using a reset or normalizer is optional if you don't mind a little variation in how your page is displayed. You may be able to address inconsistencies on a per-case basis. That said, let's look at each "clean slate" method.

CSS Resets

The older approach is a **CSS reset**, a collection of style rules that overrides *all* user agent styles and creates a starting point that is as neutral as possible. With this method, you need to specify all the font and spacing properties for every element you use. It's a truly from-scratch starting point.

The most popular reset was written by Eric Meyer (the author of too many CSS books to list) in 2007. You can find it on the web at ***meyerweb.com/eric/tools/css/reset/*** (see **Note**), and I've included a copy of it with the materials for this chapter for your copy-and-paste pleasure. I'll also present it here so you

NOTE

I also recommend reading Eric's posts about the thinking that went into his settings at ***meyerweb.com/eric/tools/css/reset/*** *and* ***meyerweb.com/eric/thoughts/2007/04/18/reset-reasoning/****.*

can get an idea of what it does. If you look through the code, you'll see that the margins, border, and padding have been set to 0 for a long list of block elements. There are also styles that get typography to a neutral starting point, clear out styles on lists, and prevent browsers from adding quotation marks to quotes and blockquotes:

```
/* http://meyerweb.com/eric/tools/css/reset/
v2.0 | 20110126 License: none (public domain)*/
html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center, dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,figure, figcaption, footer,
header, hgroup,menu, nav, output, ruby, section, summary,
time, mark, audio, video {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    font: inherit;
    vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,footer, header, hgroup,
menu, nav, section
  {display: block;}
body {line-height: 1;}
ol, ul {list-style: none;}
blockquote, q {quotes: none;}
blockquote:before, blockquote:after, q:before, q:after {
    content: ''; content: none;}
table {border-collapse: collapse; border-spacing: 0;}
```

To use the reset, place these styles at the top of your style sheet so your own styles override them. You can use them exactly as you see them here or customize them as your project requires.

Although it's a classic and still works, Eric's reset is getting on in years. Josh Comeau has published an updated CSS reset that addresses a wide range of issues with fewer lines of code. You can read about it in Josh's article "A Modern CSS Reset" at *joshwcomeau.com/css/custom-css-reset*.

Normalizers

A more nuanced approach is to use a **normalizer**, which is less drastic than a complete reset in that it maintains some styles that are useful and consistent across browsers (for example, heading margins). Normalizers attempt to smooth out differences between browsers' default styles rather than to erase them completely.

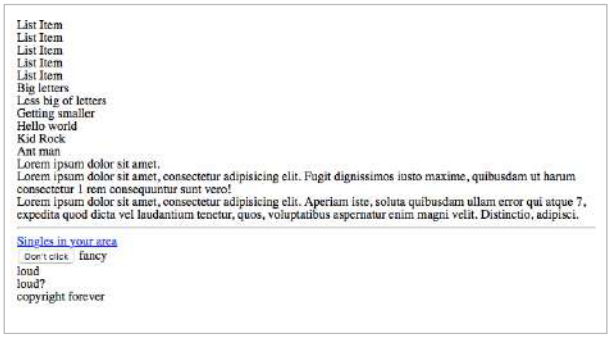
The original and still most popular normalizer is Normalize.css, created by Nicolas Gallagher and Jonathan Neal. They painstakingly combed through the user agent styles of every modern browser (desktop and mobile) and created a style sheet that tweaks their styles for consistency, rather than just turning them all off. Normalize.css gives you a reasonable starting point: paragraphs still have some space above and below, headings are bold in descending sizes, lists have markers and indents as you would expect. It also makes form widgets consistent, which is a nice service. FIGURE 21-4 shows the difference between the CSS reset and Normalize.css starting points.

You can download Normalize.css at ***necolas.github.io/normalize.css*** and include it before your own styles. It's too long to print here, but you will find that it is well organized and includes comments with clear explanations for each section.

There are lots of normalizers out there, and they have more variation than resets because their creators make decisions about what styles to tweak and what to keep intact. You may also like the shorter code and up-to-date features of Sindre Sorhus's "modern-normalize," which uses Normalize.css as a starting point (***oreil.ly/O0bHv***). If those don't suit you, there are over a hundred other options (as of this writing) available on GitHub at ***github.com/topics/css-reset***.

Normalizers are considered a superior successor to cruder CSS resets, but I think it is important to be aware of both options. Or, if slight differences from browser to browser are just fine with you (as they are for a lot of professional developers), you don't need to use either.

CSS reset


List Item
List Item
List Item
List Item
List Item
Big letters
Less big of letters
Getting smaller
Hello world
Kid Rock
Ant man
Lorem ipsum dolor sit amet.
Lorem ipsum dolor sit amet, consectetur adipisicing elit. Fugit dignissimos iusto maxime, quibusdam ut harum consectetur 1 rem consequuntur sunt vero!
Lorem ipsum dolor sit amet, consectetur adipisicing elit. Aperiam iste, soluta quibusdam ullam error qui atque 7, expedita quod dicta vel laudantium tenetur, quos, voluptatibus aspernatur enim magni velit. Distinctio, adipisci.
Singles in your area
Don't click fancy
loud
loud?
copyright forever

Normalize.css

List Item
List Item
List Item
List Item
List Item
Big letters
Less big of letters
Getting smaller
Hello world
Kid Rock
Ant man
Lorem ipsum dolor sit amet.
Lorem ipsum dolor sit amet, consectetur adipisicing elit. Fugit dignissimos iusto maxime, quibusdam ut harum consectetur 1 rem consequuntur sunt vero!
Lorem ipsum dolor sit amet, consectetur adipisicing elit. Aperiam iste, soluta quibusdam ullam error qui atque 7, expedita quod dicta vel laudantium tenetur, quos, voluptatibus aspernatur enim magni velit. Distinctio, adipisci.
Singles in your area
Don't click fancy
loud
loud?
copyright forever

FIGURE 21-4. The difference between CSS reset (left) and Normalize.css (right). (Credit: Pen created by Zach Wolf [*oreil.ly/pXM1r*])

SUPPORTING OLDER BROWSERS (CSS FEATURE DETECTION)

Dealing with uneven browser support is an inevitable challenge for web developers and designers. Although the pace of feature adoption is faster now than ever, new web technologies and CSS features are also rolling out at an accelerating rate. Unfortunately, older browsers that don't support them don't fade into extinction as quickly as we may like.

In this section, we'll look at the browsers that developers typically take into consideration, which CSS features are safe to use, and some strategies for providing fallbacks to non-supporting browsers, with a focus on CSS feature detection.

Deciding Which Old Browsers to Support

OK, let's just get Internet Explorer out of the way. Once the dominant browser, Microsoft laid it to rest a decade ago and replaced it with Edge. Because global usage is currently less than 1% for all versions of IE combined, the general practice is not to jump through extra hoops to support it. However, because it was tied to certain Windows operating system releases, and because some offices and communities are tied to those Windows operating systems, it is not 100% dead. If the browser statistics for your site show that

Glimpse the Future with Feature Flags and Nightly Builds

Browser makers start tinkering with cutting-edge web technologies behind the scenes before the specs are fully settled. Many of these experimental features are released behind a **feature flag**, which means they are not available in the browser by default, but you can turn them on manually. This gives developers a chance to start playing around with and testing the latewst CSS properties and other features before they are ready to release to the public. Here's how to turn on feature flags in the major browsers:

Chrome: Go to ***chrome://flags***.

Safari: Select **Develop → Feature Flags**.

Edge: Go to ***edge://flags***.

Firefox: Go to ***about:config*** to get a very long list of features that you can toggle on or off. The CSS-related features start with *layout.css*.

You can also download versions of browsers aimed at developers to try out the very latest features. The major browsers offer Beta, Developer, and "nightly build" versions (from most to least stable, respectively). Chrome's nightly build release is called Chrome Canary (***google.com/chrome/canary/***). Edge also has a Canary release (it uses the same Blink rendering engine as Chrome), available at ***oreil.ly/KIG_K***. Apple offers a Safari Technology Preview (***oreil.ly/nk_AY***), and Firefox has links to its developer releases at ***oreil.ly/fR9hc***.

your audience is likely to use IE, or if you have so many millions of users that 1% amounts to significant usage, providing fallbacks to support it may be worth the extra effort.

Today's Big Four browsers (Chrome, Safari, Firefox, Edge, and their mobile counterparts) get updated automatically, so it is less likely that users will be using antiquated versions. On mobile, browser updates may be tied to operating system updates (such as Safari on iOS), which don't happen as frequently. If you are using cutting-edge CSS features that just gained support in the last couple of years, you should consider fallbacks for the major browsers as well.

Deciding Which Features Are OK to Use

Thankfully, we have some good resources to guide our decisions and strategies for taking advantage of new features. As a web designer and developer, you should be familiar with these projects:

***Can I Use* (*CanIUse.com*)**

The hands-down most useful resource for evaluating the reach of CSS and other web technology features is the CanIUse site, which I've referenced many times throughout this book. When perusing browser support, you may find that some properties are supported using "vendor prefixes." That is a system browsers once used to release new CSS properties. If you need to support old browser versions, you may still need to prefix certain properties. The **"Vendor Prefixes"** sidebar has more information.

***Baseline* (*web.dev/baseline*)**

Created by the team behind the Chrome browser, the "Web Platform Baseline" (a.k.a. Baseline) project aims to provide clear information about which web features are safe to use. Baseline identifies two milestones: "newly available," when a feature is supported across the Big Four browsers and their mobile versions (the "core" browser set), and "widely available," after 30 months of core support, when the feature "can be used by most sites without worrying about support." Baseline status is now indicated for features in CanIUse, on MDN Web Docs, and on web.dev. If your management team is saying "you can't use that yet," Baseline is the authoritative reference to point them to.

***The Interop Project* (*github.com/web-platform-tests/interop/*)**

The Safari/WebKit team also keeps track of which cutting-edge features are supported by the Big Four. Their Interop Project aims to encourage the browser engines to focus on supporting features in specific areas as requested by the developer community. They publish the results of their Web Platform Tests in yearly benchmarks, the most recent of which as of this writing is Interop 2024 (*wpt.fyi/interop-2024*). Jen Simmons's blog post provides useful background on the project and progress made (*oreil.ly/NE5rP*).

Vendor Prefixes

As described in the sidebar **"Glimpse the Future with Feature Flags and Nightly Builds," t**hese days, browsers hide experimental features behind feature flags (options you can turn on or off) or in separate technology preview releases that developers can access for testing purposes only.

From the early '00s until around 2018, they kept their experimentation separate from the final implementation by adding a **vendor prefix** (or **browser prefix**) to the property or function name. The prefix indicates that the implementation is proprietary and still a work in progress. For example, while Safari was implementing text-wrap shapes, it used its own **`-webkit-`** prefixed version of the standard **`shape-outside`** property:

```
-webkit-shape-outside: url(cube.png);
```

TABLE 13-1 lists the prefixes used by the major browsers.

TABLE 13-1. Browser vendor prefixes

Prefix	Organization	Most popular browsers
`-webkit-`	Originally Apple; now open source	Safari, Chrome, Android, many others
`-ms-`	Microsoft	Internet Explorer
`-moz-`	Mozilla Foundation	Firefox, SeaMonkey
`-o-`	Opera Software	Opera, Opera Mini, Opera Mobile

Take a look at the code we used to write to ensure browser support for a simple linear gradient (just the last line will suffice today):

```
#box {
  background: #ffff00; /* solid color for old browsers */
  background: -moz-linear-gradient(top, #ffff00 0%, #00ff00 100%);
  background: -webkit-gradient(linear, left top, left bottom, →
     color-stop(0%,#ffff00), color-stop(100%,#00ff00));
  background: -webkit-linear-gradient(top, #ffff00 0%,#00ff00 100%);
  background: -o-linear-gradient(top, #ffff00 0%,#00ff00 100%);
  background: -ms-linear-gradient(top, #ffff00 0%,#00ff00 100%);
  background: linear-gradient(to bottom, #ffff00 0%,#00ff00 100%);
}
```

Vendor prefixes allowed developers to start using cool new CSS features in the browsers that supported them, which was a plus for moving web design and the specification forward. On the downside, the whole system turned out to be complicated and often misused. In the end, the browser makers agreed to put the prefix system to rest and not release prefixed properties.

However, there are a few CSS features that came into vogue during the prefix era that require prefixes in order to work in browsers released prior to 2020, should you choose to support them.

Prefixing tools

Writing all those redundant prefixed properties is a big pain, but fortunately, we have tools to generate them automatically. Autoprefixer takes CSS written in standard syntax, parses the styles, then automatically adds prefixes just for the properties and notations that need them. There is an online version that you can copy and paste your styles into at ***autoprefixer.github.io***. Prefixing is also built into CSS preprocessors like Sass and Less.

It is important that you have some familiarity with vendor prefixing, but truthfully, prefixes will likely only be useful for edge cases and eventually will not be needed at all.

Browser Support Strategies

Developers deal with non-supporting browsers in a number of ways, including progressive enhancement, JavaScript polyfills, and CSS feature detection.

The most straightforward approach is to use good ol' progressive enhancement, in which you create a reliable baseline experience for all browsers and then layer on riskier, nice-to-have features for browsers that support them. That certainly beats avoiding new features entirely while waiting for browsers to catch up.

Another approach is to *make* the browser support a new feature by using a polyfill. A **polyfill** is a set of JavaScript instructions that adds functionality to browsers, including support for HTML, CSS, and other web features. For example, if you want to use container queries but are concerned that your layout will break on older browsers, you could include a container query polyfill that tells the browser how to handle them correctly. Bear in mind, however, that it is not a given that JavaScript itself will be available, so progressive enhancement still comes into play. Polyfills are still used today, but they are far less common than they used to be when Internet Explorer was still in heavy circulation.

> **FURTHER READING**
>
> Jay Hoffman wrote a nice little history of polyfills for his History of the Web project. See "Filling Gaps with a Polyfill" (*thehistoryoftheweb.com/filling-gaps-with-a-polyfill/*).

The current best practice for dealing with older browsers is to give them fallback styles with CSS feature detection. This deserves its own section.

CSS Feature Queries (@supports)

Feature detection allows us to take advantage of cutting-edge CSS while also providing thoughtful fallbacks for non-supporting browsers. CSS feature queries can't test for HTML or JavaScript features, so other approaches are still required for those technologies.

To test for a feature, use an `@supports` rule that checks for browser support of a particular property and value declaration (see **Note**). Commonly referred to as a **feature query**, it works like a media or container query in that it runs a test, and if the browser passes that test, it applies the styles contained in the curly braces. The syntax for `@supports` is as follows:

```
@supports (property: value) {
  /* Style rules for supporting browsers here */
}
```

NOTE

The browser has to report for itself whether it has implemented the feature. If the feature is implemented in a buggy way, you may still encounter problems even when using feature queries.

The query is for an entire declaration, both the property and a value (note that there is no semicolon at the end). It was designed this way because sometimes you may test for a new property (such as `initial-letter`), and sometimes you may need to test for a new value for an existing property. For example, the `display` property is universally supported, but the newer `grid` keyword value may not be.

Let's look at a more specific example. I'd like to resize the `h1` elements' font based on the width of their containing "listing" elements by using container

queries. However, because container queries are relatively new as of this writing, it is recommended to provide fallback styles as well.

I've decided that 1.5rem will look pretty good as a fallback for everyone, so I've provided that style rule first. Next, the **@supports** rule tests for container query support by testing for **container-type: inline-size** and provides all the styles related to container queries.

THE MARKUP

```
<div class="listing">
  <div class="content">
    <h1>This is a headline</h1>
    <p>This is more content...</p>
  </div>
</div>
```

THE STYLES

```
.content h1 {
  font-size: 1.5rem;
}

@supports (container-type: inline-size) {

  /* container query styles */
  .listing {
      container-type: inline-size;
  }
  @container (inline-size < 540px) {
      .content h1 {
          font-size: 1.25rem;
      }
  }
  @container (inline-size > 800px) {
      .content h1 {
          font-size: 2.5rem;
      }
  }
}
```

Keep in mind that you do not need to use a feature query every time you want to use a newer CSS property. Browsers just ignore properties they don't recognize, so you can supply a fallback rule followed by the preferred rule directly in the style sheet. Or, you may decide that if a fancy property isn't supported (such as the **initial-letter** property for making the first letter of a paragraph decorative), the page works perfectly fine without it entirely. Feature queries are especially helpful for keeping all the rules related to a specific feature separate from the baseline styles.

Operators

The **@supports** rule can be used with three operators to refine the feature test: **not**, **and**, and **or**. When using operators, each property/value pair goes in parentheses. Let's take a look at how they work:

■ FURTHER READING

Read about feature queries in CSS Conditional Rules Module Level 3 (*w3.org/TR/css-conditional-3*). The Level 4 module (*w3.org/TR/css-conditional-4*) adds functions that test for selectors, font types, and font technologies. These are all summarized succinctly in the MDN Web Docs **@supports** reference (*oreil.ly/zMPrG*).

not

The **not** operator lets us test for when a specific property/value pair is *not* supported:

```
@supports not (mix-blend-mode: hue) {
  /* styles for non-supporting browsers */
}
```

Logically, this would be all you need to supply fallback styles, but because the older browsers you are providing fallbacks for might not recognize **@supports** rules either, you risk them skipping the rule entirely. Setting the fallbacks and overriding with styles for supporting browsers is a safer way to go for most scenarios.

and

The **and** operator applies styles only when all of the conditions in a series of two or more are met. For example:

```
@supports (border-radius: 10em) and (shape-outside: circle()) {
  /* styles that apply only when the browser supports
     shape-outside AND border-radius */
}
```

or

Use the **or** operator to apply styles when any of a series of conditions are met. This example shows it used for similar vendor-prefixed properties:

```
@supports (-webkit-transform: rotate(10deg)) or
          (-ms-transform: rotate(10deg)) or
          (transform: rotate(10deg))
  /* transform styles */
}
```

WARNING

Although you aren't likely to encounter it much today, it is theoretically possible for a browser to support a feature but not recognize **@supports**. *The classic example from back in the day was that Safari 8 supported Flexbox but not feature queries, so any Flexbox rules placed in an* **@supports** *rule would be ignored. That scenario isn't likely to impact your work, but it is worth noting that browser support for* **@supports**, *although widespread, is not universal.*

CSS CUSTOM PROPERTIES (VARIABLES)

It's common for certain property values to get used over and over again in a site design. For example, a company's brand color may appear hundreds of times in a complex style sheet. Not only is that repetitive, but if the color value changes, it requires hundreds of edits.

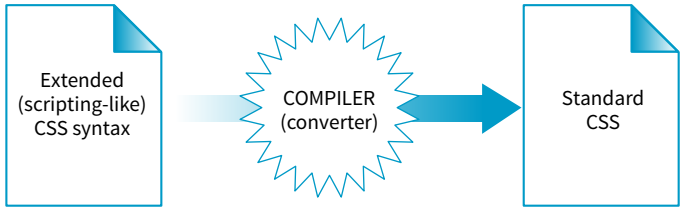
CSS custom properties (more often referred to as **CSS variables**) make handling repeated values more efficient. **Variables**—values you define once and pull in when needed—have long been a feature in traditional programming languages (see **Note**). Variable functionality was first introduced to the CSS workflow by the CSS preprocessor Sass (see the **"CSS Preprocessors (Sass and Less)"** sidebar). Variables proved to be so useful and popular that the W3C got to work adding them to the CSS standard with CSS Custom Properties for Cascading Variables Module Level 1 (***w3.org/TR/css-variables-1***), released in 2017. Now you can use variables with CSS in a way that is integrated with the DOM and allows functionality not possible with preprocessors.

NOTE

CSS variables do not behave exactly like variables in other programming languages. They are computed as needed and not stored and reused.

CSS Preprocessors (Sass and Less)

Preprocessors are languages that let authors use the time-saving syntax conventions of traditional programming languages when writing CSS. They are made up of two components: an authoring syntax and a program that translates (or **compiles**, to use the proper term) files written in that syntax to plain old CSS files that browsers understand (FIGURE 21-5).



.scss	Sass Ruby gem or Node Sass (JS)	
.less	less.js	*.css*
.styl	stylus.js	

FIGURE 21-5. A simplified view of the role of a preprocessor

By far the most popular preprocessor is Sass (**Syntactically Awesome Style Sheets**), created by Hampton Catlin and Nathan Weizenbaum, who were tired of the repetitiveness of normal CSS. A compiler is required to convert the syntax used by Sass to standard CSS syntax, and it saves the resulting file with the *.css* suffix. Other preprocessors, such as Less (Leaner Style Sheets) and Stylus, work generally the same way, but they have slightly different syntaxes and use different compilers. All of these tools are installed and run via the command line.

When preprocessors first came along, they offered efficiencies that were not possible with standard CSS, including:

- Variables that made it possible to reuse values
- Nested style rules for writing rules matching the structure of the document, thus eliminating repeated selectors
- Mixins, a convention for saving and reusing sets of styles (for example, a specific combination of background, color, and border styles)
- Loops and conditionals (such as `@if` and `@else` rules) that allow authors to test for conditions, making writing CSS closer to writing a scripting language

Variables and nested style rules are now part of standard CSS, and because they are part of the DOM, they take part in the cascade and can be accessed by JavaScript. Documents written in preprocessor languages get translated to standard CSS right before the page displays, so you miss out on those benefits.

However, Sass, Less, and other preprocessors are still central to the development workflow in many organizations and are worth exploring if you intend to work as a developer. The following links should get you started:

- Sass (***sass-lang.com***)
- Sass tutorial by Geeks for Geeks (***geeksforgeeks.org/sass***)
- Less (***lesscss.org***)
- Stylus (***stylus-lang.com***)

Custom properties are easy to use. Two steps are required:

1. Name (declare) the custom property and give it a value.
2. Use the named value with the **`var()`** function in a style declaration.

This simple example should give you a basic mental model for how it works. A primary UI color (a dark green with the hex value **`#006611`**) will be used repeatedly in my style sheet, so it makes sense to set it as a variable. First, I declare a custom property with the name **`--ui-primary`** and assign it the value **`#006611`**. The two dashes (`--`) indicate that this is a custom property. Declaring it in a rule with **`html`** as the selector makes this variable inherit to all elements in the document:

`--name` is the custom property.

`var(--name)` is a function that uses the custom property value.

```
html {
  --ui-primary: #006611;
}
```

Whenever I want to use that color value, instead of writing the hex value, I use the **`var()`** function with the custom property name:

```
button {
  background-color: var(--ui-primary);
}
a:link {
  color: var(--ui-primary);
}
```

When the page renders, the effect is the same as:

```
button {
  background-color: #006611;
}
a:link {
  color: #006611;
}
```

Now if I change the primary UI color to orange, I can update the value in the custom property instead of in every place it's used. We'll get into more detail on each of these steps in a moment, but first I want to point out a few features that are not apparent just by looking at this example:

- Because this is standard CSS, variables take part in the normal cascade, so they inherit, override, and have specificity calculated according to the standard rules. You can use inheritance strategically and also limit which parts of the document can use the variable. (I'll touch on this briefly in the upcoming discussion on scope.)
- Variables are part of the DOM, which means they can be manipulated with JavaScript. For example, you could allow a user to choose their preferred color scheme (say, a "light" or "dark" scheme) and feed their choice to the custom property dynamically with a script. Using custom properties with JavaScript is outside the scope of this chapter, but it's good to know it is an option.
- Names are more intuitive than numeric values, so using variables can improve the semantics of the style sheet.

Declaring the Custom Property

As you've seen, it's pretty easy to set up a variable name by prefixing it with two dashes (e.g., `--ui-primary`), but there are a few more things you should know. First, the custom property must be defined in the declaration block of a rule with a valid selector. In other words, it can't be hanging out in the style sheet on its own. Also, it's best to stick with letters, numbers, and dashes when naming to ensure there are no problems with special characters. Finally, names are case-sensitive, so `--brand-color` is not the same as `--Brand-Color`.

The element you apply the custom property to is important too. You can make variables apply to the whole document (**globally**) or to a specific element and its descendants (**locally**), depending on where you declare the custom property. This is known as setting its **scope**. If you want the variable to apply globally, define it at the root of the document by using the `:root` selector or using the `html` element as the selector (they are the same thing, for HTML documents). Some developers use the `body` element for the same effect. In this example, any element in the document can use the `--border-strong` variable shortcut for the value of `border`:

```
:root {
  --border-strong: 2px solid #ccc;
}
```

You can also limit the scope of a variable by defining it with a more specific selector, in which case it will be available only to matching elements and their descendants. Here, the variable will be accessible only to elements with the class "alert" and their descendants:

```
.alert {
  --border-strong: 6px solid red;
}
```

Notice that I've used the same variable name in both examples. If these were in the same style sheet, borders that use the `--border-strong` variable would be 2px and `#ccc` (gray) throughout the document, except for `.alert` elements, where they would be 6px and red (because that selector has more weight). If you give a local variable the same name as a global variable (or any variable set higher in the document tree), it will override it. The cascade works the same way for custom properties as you would expect for traditional properties and values.

There is a second way to declare custom properties, using the `@property` at-rule, which is briefly discussed in the sidebar **"Registering Custom Properties with @property."**

If the browser can't find or doesn't understand a variable in a `var()` function, it ignores the style declaration entirely. Fortunately, you can provide a fallback value after a comma that the property can use if the variable doesn't work. This is especially useful if you know that the value for the variable is being provided with JavaScript, which has a potential to fail. In this example,

Registering Custom Properties with @property

Another method for creating a custom property is to use an `@property` at-rule. It "registers" a custom property with the browser's rendering engine with all the information the browser needs to implement it, such as its syntax, default value, and whether it inherits. This is similar to the way a polyfill creates browser support for a feature; however, it uses native CSS, so no JavaScript is required. To give you an idea of what this looks like, the following `@property` declaration creates a new custom property called `--jaunty-tilt` with an initial value of `-7deg` that can be used for rotation values:

```
@property --jaunty-tilt {
  syntax: '<angle>';
  initial-value: -7deg;
  inherits: false;
```

The `@property` API is a more recent addition to the spec and has only been supported by Chrome, Edge, Safari, and Opera since 2023. Firefox does not support it at all as of this writing.

Using `@property` is outside the scope of this chapter, but you can read more about it in the CSS Properties and Values API Level 1 spec (*oreil.ly/NuuqW*). For a detailed description of a use case, see "Times You Need a Custom @property Instead of a Variable" by Preethi Sam (*oreil.ly/nuj36*).

if the browser can't find the value for **`-ui-primary`**, it will use the color green instead:

```
button {
  background-color: var(--ui-primary, green);
}
```

Note that this is not a fallback for browsers that don't support custom properties. This is for browsers that know what to do with the **`var()`** function, but aren't able to use the variable name provided because it is unrecognized or undefined.

Browser Support

Speaking of browser support, custom properties defined in style rules have been well supported by the major browsers since 2016 and will work for 97% of users as of this writing.

Further Reading

Once again, there's a lot more to the story than I can fit in this section. Custom properties are a useful tool for your CSS authoring toolbelt, so to learn more, I recommend reading these resources:

- Chris Coyier's article "A Complete Guide to Custom Properties" on CSS-Tricks (***oreil.ly/VA895***) is a clear introduction and has a bunch of interesting and useful examples of how variables can be used.
- "Using CSS Custom Properties (Variables)" on MDN Web Docs (***oreil.ly/UzhXm***) is another thorough introductory walk-through.
- "How to Use CSS Variables Like a Pro" by Idorenyin Obong on the *LogRocket* blog (***oreil.ly/FJRwd***) has a number of project examples.

A QUICK INTRODUCTION TO CSS NESTING

Variables in CSS were first introduced in the Sass preprocessor language and were added to the CSS spec by popular demand soon thereafter. Another authoring feature of Sass that was popular enough to add to the spec is **nesting**. CSS nesting allows you to write selectors in a way that mimics the structure of the document, making the CSS more modular and easier to maintain while reducing repetition in selectors for potentially smaller files. This quick introduction should give you the basic idea of how nesting works.

Here is a **nav** element that contains an unordered list with several options:

```
<nav>
  <ol>
    <li>Option1</li>
    <li>Option2</li>
    <li>Option3</li>
  </ol>
</nav>
```

Traditionally, to apply styles to each of these nested elements, you would repeat the element names in each selector:

```
nav {
  /* nav styles */
}

nav ul {
  /* ul styles */
}

nav ul li {
  /* li styles */

}
```

With CSS nesting, you can nest the style rules for the **nav**, **ul**, and **li** elements inside the declaration block. The **&** symbol indicates a **nesting selector** and stands in for the higher-level selector. For example, **& ul** is the equivalent of **nav ul**. Nesting alleviates the need to type out the selectors multiple times, as you can see here:

```
nav {
  /* nav styles */

  & ul {
    /* ul styles */

    & li {
      /* li styles */
    }
  }
}
```

When the nested selectors are simple element selectors, as shown above, the spec requires the **&** symbol to be present in order for them to be valid. However, if the nested selector starts with a special selector character such as **:**, **+**, or **>**, as shown in the following example, you may omit the **&**:

```
button {
  /* button styles */

  :hover {
    /* button hover styles */
  }
  :active {
    /* button active styles */
  }
}
```

Due to inconsistent browser behavior, it is recommended that you include the `&` for all nested selectors for the time being. This also gives you more control over how the parent selector is referenced.

As of this writing, nesting is supported in the major browsers, but only recently (from mid-2023). The spec has been in flux and may continue to shift; taking that into consideration in combination with spotty browser support, nesting may not be *quite* ready for production code. Support improves quickly, so check ***CanIUse.com*** if you'd like to use it in your style sheets.

Further Reading

I've given you the general idea of how nesting works here, but of course, there is a *lot* more to it. The following resources provide more thorough explanations:

- The CSS Nesting Module (***w3.org/TR/css-nesting-1/***)
- "Using CSS Nesting" on MDN Web Docs (***oreil.ly/y_nt6***)
- Kevin Powell's video "Getting Started with CSS Nesting" (***oreil.ly/ki0XO***)
- "Native CSS Nesting: What You Need to Know" by Sarah Chima Atuonwu (***blog.logrocket.com/native-css-nesting/***)

Keep in mind that the spec may have been tweaked since I wrote this, so you may want to search for more recently published articles. Another excellent resource to consult is Frontend Dogma's "nesting" archive (***frontenddogma.com/topics/nesting/***), which provides an up-to-date list of links to articles and resources related to CSS Nesting.

AN EVEN QUICKER INTRODUCTION TO CSS LAYERS

In the "You probably don't need it now, but it's nice to know it's out there" department, we have CSS Layers. **CSS Layers** give authors more control over the cascade and which rules override others. You've learned that when there are conflicts, rules with more specific selectors override those with less specific selectors, and that rules that come later in the style sheet override those listed earlier. Used thoughtfully, that's sufficient for most scenarios. However, for complicated sites and sites with style sheets imported from third-party libraries, it's easy for rules to get overridden accidentally (or even mysteriously). CSS Layers allows you to identify sets of style rules organized by purpose, by source, or by some other criteria, and specify their order of priority.

Here's one way that may look in a style sheet. This example uses the `@layer` at-rule to list named layers of style rules in order from least important to most important. Here, styles in the "theme" layer take precedence over the "framework" layer styles, which in turn will override styles in a "reset" layer.

This declaration of layer priority should be the first thing in the style sheet. The sets of styles for each layer are declared with the **`@layer`** at-rule, followed by the layer name and the applicable styles provided in curly braces:

```
@layer reset, framework, theme;

@layer theme {
  .alert {
    border-color: green;
  }
}

@layer framework {
  .alert {
    border-color: red;
  }
}
```

In this (very short) style sheet, you can see that there are conflicting style rules for **`.alert`** elements. According to the normal cascade, the last one (red) would win. But because we've explicitly stated that the "theme" layer is more important than the "framework" layer, the **`.alert`** border color will be green. Just from this small example, you can see how layers take some of the burden of writing styles in exactly the right order off the author, because you can be assured of which style layers have the final say.

Styles not included in a layer take precedence over all styles defined in layers. If the following rule appeared in the style sheet on its own (i.e., not in an **`@layer`** rule), even if it came before the previous two rules, the **`.alert`** border color would be orange:

```
.alert { border-color: orange; }
```

That's pretty much all I'm going to say about CSS Layers, as they are not something you'll run into as a beginner. But because they are a powerful tool for controlling the cascade and avoiding hacks for boosting specificity, they are worth knowing about. The major browsers began supporting CSS Layers in 2022, so they are a fairly recent addition to the CSS workflow.

Further Reading

There's much more to know about CSS Layers, so I will refer you to these resources to explore on your own when you are ready:

- Miriam Suzanne's article "A Complete Guide to CSS Cascade Layers" on CSS-Tricks (*css-tricks.com/css-cascade-layers/*)
- The "Cascade Layers" tutorial on MDN Web Docs (*oreil.ly/F1-Jt*)
- The CSS Cascading and Inheritance Level 5 spec (*w3.org/TR/css-cascade-5/*)

CSS FRAMEWORKS

As a beginner, you are likely to write the styles for all of the elements in your pages manually and from scratch. For many sites, even commercial ones, that is a perfectly fine way to go. However, large-scale sites and sites created collaboratively by a team are more likely to take advantage of a CSS framework for site production. A **framework** provides a collection of prewritten and ready-to-use code for common UI elements such as grid structures, buttons, form elements, navigational menus, and alert pop-up windows, to name just a few. They may also include a reset stylesheet, icon sets, and other features. Starting with a framework gives you a nice head start.

Frameworks offer a number of benefits, including:

- Speeding up development because you don't need to create everything from scratch
- Leveraging established best practices for common UI elements, eliminating the need to reinvent the wheel
- Ensuring consistency due to team members drawing from the same collection of approved components and styles
- Utilizing mobile-first, responsive designs, including multicolumn page templates and flexible components right out of the box
- Making sure the code works across all browsers, with fallbacks already in place for older browsers

There are a *lot* of frameworks out there, each working a little differently. Some provide HTML templates and JavaScript in addition to style sheets. Others focus primarily on CSS, such as Tailwind CSS. Some require familiarity with additional languages such as Sass or Ruby. Some frameworks are more easily customizable than others. As of this writing, the most popular CSS frameworks include Tailwind CSS (***tailwindcss.com***), Bootstrap (***getbootstrap.com***), and Foundation (***get.foundation***), but a quick web search for "most popular CSS frameworks" will turn up lists of what's currently in favor (see **Note**). Most popular frameworks are open source and free to use.

NOTE

The frameworks listed in this section are primarily CSS-related. In **Chapter 25, Next-Level JavaScript***, you'll learn more about component-based frameworks such as React and Angular that can be used to construct whole sites.*

How CSS Frameworks Work

When working with a framework such as Bootstrap or Tailwind CSS, you style elements right in the HTML markup by using the `class` attribute with special, predefined class values called **utility classes**. Utility classes are purpose-specific and often apply just a single style rule to the element. This is best explained with examples (I'll be using examples from Tailwind CSS, but Bootstrap works similarly).

Here, I've made the background of a `div` white by using the `.bg-white` utility class:

```
<div class="bg-white">...</div>
```

And that's all I need to do, because the style for that class is already written. In the Tailwind style sheets, there is a rule that gives any element with the **bg-white** class value a white background color:

```
.bg-white {
  background-color: white;
}
```

Typically, elements end up with multiple utility class names, each applying one aspect of their style—for example:

```
<div class="flex w-48 p-4 bg-black text-white text-base
font-bold">...</div>
```

The result of that will be the same as if I had written the following declaration block for the **div** (note that length units in Tailwind equal .25rem):

```
div {
  display: flex;
  width: 12rem;
  padding: 1rem;
  background-color: black;
  color: white;
  font-size: 1rem;
  font-weight: bold;
}
```

It looks a little strange at first to see style information in the markup after learning that HTML is all about semantics, but keep in mind it is still external style sheets that are doing the styling here. Styling with utility classes offers some benefits:

- You sidestep the unpredictability that might arise from the cascade, such as accidentally overriding previous styles. The specificity is said to be "flat" (all selectors have the same weight).
- Utility classes are customizable, so you can assign colors, fonts, and so on for each project. The custom styles for each component are the sort of thing you would store and document in a design system (discussed in **Chapter 19, Responsive Web Design**). Note that some frameworks are more customizable than others.
- They provide a finite list of styles to choose from, which helps keep teams of developers from writing conflicting styles.

On the downside, long strings of class values can be difficult to read. In addition, you need to memorize all of the special utility class names of the framework you are using, although most are fairly intuitive and it gets easier with practice.

That's all I'm going to say about frameworks, because my job in this book is to teach you how to write CSS manually and give you a solid foundation for learning more. However, it's important that you know what they are and what they are good for. If you work in web design professionally, there is a good chance you and your team will use some sort of framework in the production workflow.

WRAPPING UP STYLE SHEETS

That concludes our (long, long) tour of Cascading Style Sheets. You've come a long way since styling an `h1` and a `p` back in **Chapter 11, Introducing Cascading Style Sheets**. By now, you should be getting comfortable formatting text and even doing basic page layout. While CSS is easy to learn, it takes a lot of time and practice to master. If you get stuck, you will find that there are many resources online to help you find the answers you need. The nice thing about CSS is that you can start with just the basics and then build on that knowledge as you gain proficiency in your web development skills. If I've done my job right, you should have a good CSS foundation.

In **Part IV, JavaScript for Behavior**, I hand over the keyboard to Aaron Gustafson, who be your guide to learning JavaScript. I'll be back in **Part V, Web Images**.

TEST YOURSELF

See how well you picked up the CSS techniques in this chapter with these questions. As you may have guessed, the answers are available in **Appendix A**.

1. What is the purpose of a CSS reset?
   a. To override browser defaults
   b. To make presentation more predictable across browsers
   c. To prevent elements from inheriting unexpected styles
   d. All of the above

2. Name two approaches to aligning form controls and their respective labels without tables. A general description will do here.

3. What do double dashes (`--`) indicate in a style declaration?

4. When does it make sense to use a variable in CSS?

5. Use the declared custom property as the value for padding:

```
:root {
  --spacing: 1.5rem;
}

div.card {
  padding: _____________________;
}
```

6. Why do developers use CSS preprocessors like Sass for writing CSS?
   a. To take advantage of if/else logic statements in style sheets
   b. Because they offer code writing efficiencies found in traditional programming languages
   c. Because they generate CSS code for you automatically
   d. a and b
   e. All of the above
7. What is the advantage of using standard CSS variables instead of Sass variables?
   a. Being able to target the variable with JavaScript
   b. Participation in the cascade
   c. Rendering right in the browser without requiring a compiler program
   d. a and b
   e. All of the above
8. What character indicates a nested selector?
9. What are the "core" browsers according to Chrome's Baseline project?
10. Write the following rules using nested selectors:

```
.options {
    font-weight: bold;
}
.options li {
    font-style: italic;
    margin-top: .5rem;
}
.options li a {
    text-decoration: none;
    color: hotpink;
    border-bottom: 1px solid hotpink;
}
.options li a:hover,
.options li a:focus {
    color: forestgreen;
    border-color: forestgreen;
}
```

11. Add the code necessary to make this rule apply only to browsers that support it:

```
#layout {
  display: grid;
  grid-template-columns: 1fr 2fr 1fr;
  grid-template-rows: 12rem auto ;
}
```

CSS REVIEW

Here is an at-a-glance roundup of the newer CSS features introduced in this chapter.

Feature query syntax

```
@supports (property: value) {
  /* style rules for supporting browsers here */
}
```

Custom properties (variables)

Custom property declared on an element (defines its scope):

```
scoped-selector {
  --custom-property-name: assigned-value;
}

selector {
  property: var(--custom-property-name);
}
```

Nesting

Use the **&** (ampersand) character to indicate nested styles:

```
E1 {
  /* E1 styles */

  & E2 {
    /* E1 E2 styles */
  }
}
```

CSS Layers

```
@layer comma-separated list of layer names to establish hierarchy;

@layer CSS-layer-name {
  /* styles for that named CSS layer */
}
```

IV

JAVASCRIPT FOR BEHAVIOR

CHAPTER
22

GETTING STARTED WITH JAVASCRIPT

STATEMENTS AND VARIABLES

by Aaron Gustafson

Up until now, you've been working with HTML and CSS to build web pages. They're languages that will take you far, and you can build some powerful and interesting websites with only those tools in your toolbox. At a certain point, however, you're probably going to want to create something HTML and CSS can't deliver on their own. When you hit that point, you'll need JavaScript.

JavaScript is first and foremost a programming language created to add interactivity to web pages. Want to validate a form? You can use JavaScript. Create a real-time chat application? JavaScript is your go-to. Read and write to a user's cookies? JavaScript can do that. Load more content as a user scrolls down the page? JavaScript has you covered. Gather real-time data about how users are interacting with your pages? Yup, JavaScript can handle that too.

JavaScript is an immensely popular language. It's an integral part of every major website and powers a bunch of popular desktop apps as well, including Adobe Photoshop, Slack, and Figma. It's even used to run the UI astronauts have used to control spaceships! Developers who are skilled at JavaScript are in high demand across the globe.

What's in a Name?

Given its name, you might think JavaScript is related to Java, another programming language. It's not. The "Java" in "JavaScript" is a nod to the popularity of the Java language at the time JavaScript was invented. That's where their similarities end.

What we affectionally call "JavaScript" isn't even technically "JavaScript" anymore; it's ECMAScript. Ecma International, the standards organization that oversees it, defines the language in the ECMA-262 specification. JavaScript remains its common name, though, so that's how I'll refer to it. It's worth noting that some people shorten JavaScript to "JS," but for clarity I'll use the full name.

IN THIS CHAPTER

A simple script

Writing statements

Working with numbers

Working with text strings

Defining and using variables

Embedded versus external JavaScript

Using the JavaScript console

If you have no prior experience with programming, the learning curve for JavaScript may be a bit steep. It would take many more book pages than I have here to teach you all the ins and outs of the language and its ecosystem, but my goal is simply to introduce you to the most important concepts that you'll need to get started. By the end of **Part IV, JavaScript for Behavior**, you should have a basic foundation in JavaScript syntax and how it's used for common tasks on web pages. If you want to go deeper in a particular area, I'll provide resources along the way.

Now that you know a little more about what JavaScript is and does, let's take a look at a simple script.

MEET A SCRIPT

The following little script finds a paragraph with the **`id`** value "intro" (**`p#intro`**) and changes its content (which is no content, as you can see by the **`p`** element in the example) to the phrase "Hi from JavaScript!" If you are having a hard time following the code, don't worry; I'll explain each piece momentarily. This script has just enough going on to give us an opportunity to talk about some common JavaScript syntax conventions:

```
<p id="intro"></p>
<script>
  // Add text to p#intro
  document.querySelector( "p#intro" ).innerText = "Hi from JavaScript!";
</script>
```

Let's break down what's happening here:

Learning JavaScript

The JavaScript standard is massive, a testament to both its popularity and its adaptability. Many of the features that have contributed to JavaScript's growth originated within the standards bodies like Ecma International, W3C, and WHATWG. Many more were influenced by the work of JavaScript developers finding solutions to common development problems.

We will only be able to scratch the surface in this book. To really get a sense of what JavaScript can do, I recommend picking up a copy of *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly). If web references are more your thing, you can't go wrong with MDN's Web Docs (***developer.mozilla.org***). And as I mentioned, I will be providing additional resources throughout the chapters.

Another way I recommend learning what's available in JavaScript is through experimentation. Later in this chapter you'll get familiar with the JavaScript console built into your browser. You can type your code directly into the console to see how it works and get alerts about errors. One amazing feature of the console is that it makes autocomplete suggestions for you. For example, if you enter a text string and follow it with a period (a.k.a., a dot), you'll get a complete view into the properties and methods available for working with strings. From there you can play around and see what they do or you can search for the property or method on MDN for a primer.

- First, I added the script with the **`script`** element. In this example, the actual code appears between the opening and closing **`script`** tags. (The **`script`** element can also be used to reference JavaScript in a separate file with a *.js* extension, which we'll discuss later.)

NOTE

The JavaScript code needs to appear after the markup it applies to in the document. That's because the JavaScript it contains will run as soon as it's read by the browser. In this example, the paragraph comes immediately before the **`script`** *element, so we're good. If the paragraph came after the* **`script`** *element, however, the browser wouldn't have been able to find it, and nothing would have happened.*

- Inside the **`script`** element, I left a comment for myself using two forward slashes (**`//`**) (see the upcoming **"Commenting Your Code"** sidebar for more information about comments).
- Next, I wrote a JavaScript statement. A **statement** tells the browser to do something. I used a semicolon (**`;`**) to indicate the end of this statement, just like I used a period to indicate the end of this sentence. This script has only one statement that tells the browser—technically, the JavaScript engine within the browser—to inject some content into the page. Specifically, it looks in the HTML document (**`document`**) for an element that matches the CSS selector "p#intro" (**`querySelector("p#intro")`**), and changes its text content (**`innerText`**) to "Hi from JavaScript!"

Commenting Your Code

As you start writing JavaScript programs, it's a good idea to document your code with comments. They don't get shown to users, so they're a great way to leave notes that tell your team (or remind your future self) what's happening in your code or why it does something a particular way.

JavaScript supports two types of comments: **line comments** and **block comments**. Line comments start with two forward slashes (**`//`**), turning the rest of the line into a comment:

```
// This is a one-line comment
```

Block comments can span multiple lines, or just one. They start with **`/*`** and end with **`*/`**:

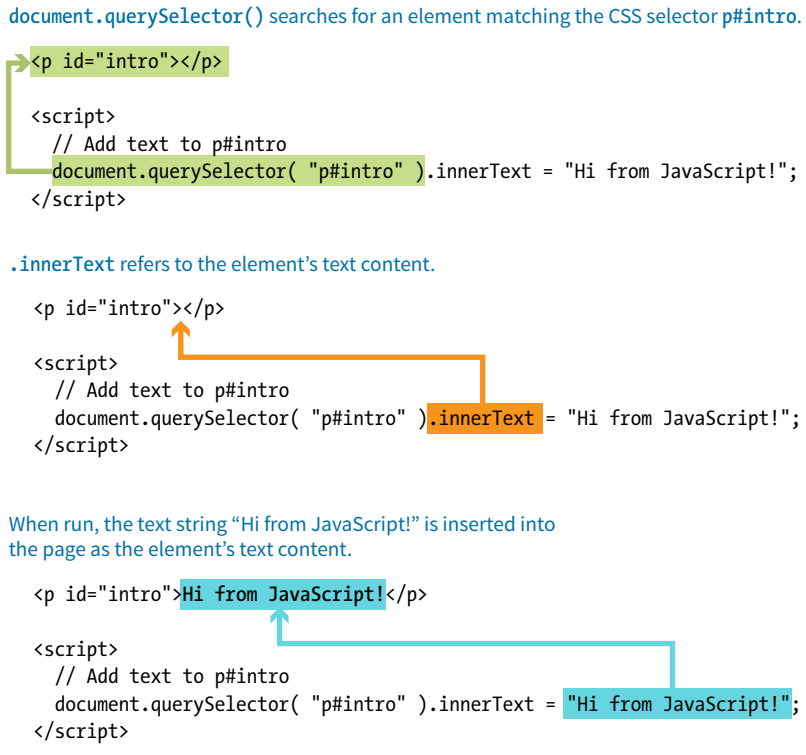
```
/* This is a block comment
   and can span multiple lines */
```

Block comments work the same way as CSS comments, making the syntax easy to remember. Here are some good ways to use comments:

- Outlining the steps needed to complete the program before you start writing code
- Documenting what a particular line of code does
- Noting why a line of code is written a particular way
- Leaving a "TODO" note to remind yourself to return to a specific part later
- Documenting the interface of a given function or method

NOTE

Throughout the JavaScript chapters, I will be using comments to indicate the result of a script or to describe what a particular statement does. So, when you see a comment in an example, it's me giving you a little inside information.


document.querySelector() searches for an element matching the CSS selector p#intro.

<script>
// Add text to p#intro
document.querySelector("p#intro").innerText = "Hi from JavaScript!";
</script>
.innerText refers to the element's text content.

<script>
// Add text to p#intro
document.querySelector("p#intro").innerText = "Hi from JavaScript!";
</script>
When run, the text string "Hi from JavaScript!" is inserted into the page as the element's text content.
Hi from JavaScript!
<script>
// Add text to p#intro
document.querySelector("p#intro").innerText = "Hi from JavaScript!";
</script>

FIGURE 22-1. Visualizing the different parts of the JavaScript code that find or act upon a part of the HTML. First, in green, `document.querySelector()` searches for an element matching the CSS selector `p#intro`. Then, `.innerText` refers to the element's text content. When the script is run, the text "Hi from JavaScript!" is inserted into the HTML as the element's text content.

FIGURE 22-1 shows how the script works on the `p` element in the example.

The single statement in our little script also demonstrates a few key concepts we'll be discussing in this chapter:

Dot notation

Dot notation is how we access functionality on an object. It uses dots (a.k.a. periods) to indicate features nested within other features. `document.querySelector()` is an example of dot notation. The `.innerText` piece after it is another. See the sidebar **"Objects and Dot Notation"** to dig in a bit more.

Functions

Functions are discrete pieces of code that execute a single task. Our code uses one function: `querySelector()`. Technically, `querySelector()` is a special kind of function called a "method" because it's attached (via dot notation) to something else: the `document` object. We'll talk more about functions and methods in upcoming chapters.

Operators

Operators are special symbols that perform operations such as calculations (like addition or subtraction), comparisons (such as more-than or less-than), or value assignments. In this example, the equals sign (`=`) assigns a value (the text phrase) to the `innerText` property of the paragraph.

Camel case

The properties and methods in JavaScript use a capitalization convention called **camel case** in which the first letter of every word, except for the first word, is capitalized (for example, `querySelector()` and `innerText`). This convention is used throughout JavaScript for function and property names. Many developers use it for variable and function names as well.

Strings

The text inside quotation marks is an example of a **string**, which is a value type made up of any number of characters. Strings are one of several kinds of information you can work with in JavaScript. We'll look at them in more detail later in this chapter.

OK, it's your turn to write your first JavaScript in EXERCISE 22-1!

Objects and Dot Notation

As you read this, you are probably touching a physical object. Maybe it's the physical version of this book. Maybe it's a tablet or ereader of some sort. Or maybe your hands are resting on a keyboard, mouse, or trackpad as you're using a computer to read these words. Regardless of what it is, it's an object. It has attributes, such as its physical dimensions, weight, and color. It also has things it can do: a book can be opened and closed, a device can be turned on and off, and so on.

JavaScript has objects too. In fact, nearly everything you work with in JavaScript is an object. In our example script, `document` is an object that is a reference to the HTML page. Just like real-world objects, JavaScript objects have attributes and things they can do. We call the attributes **properties** and the things they can do **methods**.

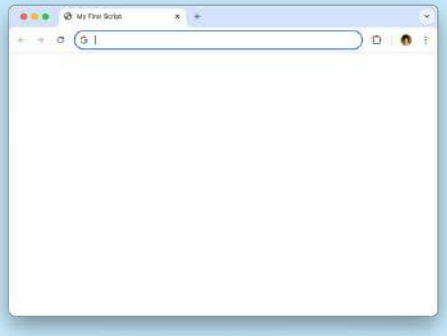
The most common way developers access the properties and methods of a JavaScript object is to use **dot notation**. The "dot" in dot notation refers to the period symbol (`.`). When you use a period in JavaScript, you're saying the bit after the period *is a part of* the bit before the period. So, for example, if I wanted to find out what character set is used on a page, I would refer to the `document` object's `charset` property, like this (see **Note**):

```
document.charset; // Likely "utf-8"
```

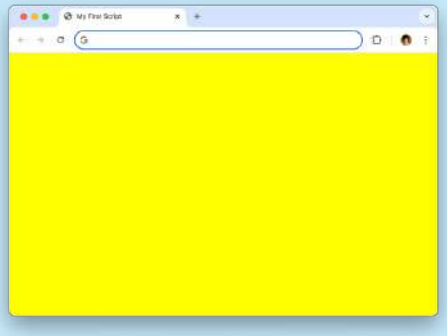
Similarly, if I want to perform an activity the `document` object provides, I can use dot notation to access one of its methods. You saw this in the example script when I called the `document` object's `querySelector()` method:

```
document.querySelector( "p#intro" );
```

For more on objects, dot notation, and more, see the MDN Web Docs article "JavaScript Object Basics" (*oreil.ly/MxXls*).



Before



After

FIGURE 22-2. The sample web page before and after running your first script

EXERCISE 22-1. Your first script

In this simple exercise, you'll get a feel for adding a **script** element to a document and seing how it alters the page when it displays.

1. Start by opening the file *color-changer.html* (provided with the materials for this chapter) in a text or code editor. You will see that it contains only the minimal structural elements (**html**, **head**, **title**, and **body**) and no content. Note also that there are no CSS styles in the source document. If you open the document in a browser, you should see a blank white screen. We are going to use JavaScript to apply a new background color to the page.
2. Insert a new **script** element inside the **body**:

```
<body>
  <script>
    // Your JavaScript will go here
  </script>
</body>
```

3. Within the **script** element, replace the comment with the following statement:

```
document.body.style = "background: yellow";
```

4. Save the file and open it in your browser. The background of the page's **body** should now be yellow (FIGURE 22-2)! If it isn't, double-check that you typed exactly what you see in the code sample in step 2, including all the punctuation.

Congrats! You just wrote some JavaScript. Hang on to this file, because you will be adding to it in upcoming exercises.

With your first script under your belt, let's talk a little more about statements.

STATEMENTS

As I mentioned earlier, a statement is an instruction. It tells the JavaScript engine executing your code what it should do. We generally write statements on individual lines, as it makes the code a lot more readable:

```
statement1;
statement2;
```

Most statements end with a semicolon. Technically, you can omit the semicolon when you write a statement on its own line, as the **newline character** (a hidden character that is added when you hit the Enter/Return key) also ends a statement. Some developers do this, but I recommend always including them because it makes it crystal clear where each statement ends.

It's possible to include multiple statements on a single line, but developers don't often do this because it makes the code harder to read. If you do write multiple statements on a single line, semicolons are required:

```
statement1; statement2;
```

Although seeing multiple statements on a single line is rare (except in `for` loops, which we'll get to in **Chapter 23, Functions, Conditionals, and Loops**), it's becoming increasingly common to write a single statement on multiple lines. You don't see it often with short statements, but for longer ones it can really help with readability. Consider this statement:

```
document.querySelector("form[role=search] button");
```

That's a lot to take in as a single line. Breaking it into multiple, indented lines, however, makes it a little easier to read:

```
document
  .querySelector(
    "form[role=search] button"
  );
```

I think you'll agree that reformatting the code in this way makes it clearer than it is using a CSS selector (`form[role=search] button`) to find a `button` within the document.

As I mentioned, statements tell the computer what to do. The order you write your statements in is also the order in which they will be run by the browser. Throughout the remainder of this and the next several chapters, you'll get a better understanding of how you can use statements to bend the browser to your will. We're going to start simply, though, with some basic math.

NUMBERS AND MATH

No matter what you're looking to do with JavaScript, it's likely to involve numbers or math in some way. Want to let a reader know how long it will take to read an article? You'll use math to divide the number of words in the article by the average number of words people can read in a minute. Want to have things move around on the page as your users scroll? You'll be tracking the user's location on the scrollbar (a number) and comparing it to certain points where you want to start or end animations (math). Numbers and math are all over the place!

Numbers are one of the data types you can use in JavaScript, and they work exactly the way numbers work in the real world. This section takes a closer look at the various types of operators available in JavaScript for handling numbers in statements.

Math Operators

So what does math look like in JavaScript? Addition and subtraction are done just as you'd expect, using their respective **operators**: `+` and `-`.

When you type `10 + 10`, JavaScript will evaluate that math statement and **return** the result of the calculation. Going forward, I'll use comments to show what JavaScript would return when evaluating a statement, like this:

```
10 + 10; // 20
```

The operators for multiplication (*) and division (/) are a little different from what we're taught in school (mainly because × and ÷ aren't common characters on a keyboard):

```
10 * 10; // 100
10 / 10; // 1
```

The standard order of operations you learned in school applies in JavaScript as well:

```
( 1 + 1 ) * 2; // 4
```

Here, the operation in parentheses—1 + 1—is evaluated first, resulting in 2. The remaining operation is then 2 × 2, which is 4. TABLE 22-1 summarizes the JavaScript math operators.

TABLE 22-1. Math operators

Operator	Function	Example	Example explained
+	Addition	`a + b`	Adds **b** to **a**
-	Subtraction	`a - b`	Subtracts **b** from **a**
*	Multiplication	`a * b`	Multiplies **a** and **b**
/	Division	`a / b`	Divides **a** by **b**
**	Exponent	`a ** b`	Raises **a** to the power of **b** (a^b)
%	Remainder	`a % b`	Returns the remainder when **a** is divided by **b**

Comparison Operators

Calculations aren't the only mathematical operations you can perform in JavaScript. JavaScript also provides a set of operators that let us compare one value to another. TABLE 22-2 lists them (see **Note**).

NOTE

Not listed here are the == (equivalent to) and != (not equivalent to) comparison operators. They are trickier to use because they don't require the compared values to be the same data type. Sticking with the === and !== will be more predictable, particularly while you are still learning.

TABLE 22-2. Comparison operators in JavaScript

Operator	Function	Example	Example explained
<	Less than	`a < b`	**a** is less than **b**
<=	Less than or equal to	`a <= b`	**a** is less than or equal to **b**
===	Equal to	`a === b`	**a** is equal to **b**
!==	Not equal to	`a !== b`	**a** is not equal to **b**
>=	Greater than or equal to	`a >= b`	**a** is greater than or equal to **b**
>	Greater than	`a > b`	**a** is greater than **b**

Comparison operators are typically used with **conditional statements** (statements that run only when a specified condition is met). For example, let's say I have some code I want to run only if a user is on a narrow screen. I can use

a comparison operator inside a conditional **if** statement to do that. Here, I'm going to compare the value of **window.innerWidth**—a built-in reference to the browser's viewport width, in pixels—against the number 600 by using the "less than" comparison operator (**<**):

```
if ( window.innerWidth < 600 ) {
  // Code for a narrow screen
}
```

Taken all together, this statement is saying *if the viewport is less than 600 pixels wide, run the code inside the curly braces*. I made up the 600 pixels value for this example, so don't get hung up on that bit.

Comparison operators resolve to either "true" or "false." The values **true** and **false** are called **Booleans**. We'll discuss Boolean values and comparison operators more when we talk about conditionals in greater detail in **Chapter 23**.

Often, you'll find it necessary to compare whether two values are equal. The operator used for that is the triple equals sign (**===**). Why triple equals? Well, the equals sign (**=**) already has a purpose: assigning values, like you saw in the script at the start of this chapter. There is also a double equals operator (**==**) for comparing equivalency (which is a slightly different thing), but we won't be using that here.

Incidentally, if you want to know if two things are *not* equal, you use an exclamation point followed by two equals signs (**!==**):

```
window.innerWidth === 600; // true when the window is 600 px wide
                           // false otherwise
window.innerWidth !== 600; // false when the window is 600 px wide
                           // true otherwise
```

Again, we'll talk more about comparison operators in **Chapter 23**.

Assignment Operators

The last math operators I want to mention before we move on are **assignment operators**.

The most basic assignment operator is the equals sign (**=**), which assigns a value to a variable. We'll talk at length about variables later in the chapter, but to give a quick example, this variable declaration statement creates the variable **total** and sets its value to 0:

```
let total = 0;
```

Other assignment operators provide a shorthand for math operations used to update a variable's value. This will be clearer with an example. After the previous statement, our variable **total** has a value of 0. If we wanted to add 5 to **total**, we could do it two different ways. The long-form way would be to set **total** equal to the result of adding 5 to itself:

```
total = total + 5;
```

Alternatively, we can use an assignment operation to do the math *and* assign the new value all at once. Here's that same statement, rewritten using the **addition assignment operator** (`+=`):

```
total += 5;
```

Either way you write it, the value of `total`, which was initially 0, gets updated to 5. You'll see assignment operators frequently in code that keeps a running total, such as when calculating the cost of a shopping cart. Just to drive home how it works, here are a few more operations that act upon that same `total` value:

```
total += 10; // 15
total -= 3;  // 12
total /= 4;  // 3
```

That's right—addition isn't the only operation you can do this way. Subtraction (`-=`), multiplication (`*=`), and division (`/=`) can also be done using assignment operators.

The final assignment operators I'll share with you are for increasing (`++`) and decreasing (`--`) a value by 1:

```
let position = 10;

++position; // 11, which is equivalent to
            // position = position + 1;

--position; // 10, which is equivalent to
            // position = position - 1;
```

The `++` and `--` are known as the **incrementing** and **decrementing** operators, respectively. You'll see these used a lot in loops, where you're usually moving from item to item in a list and need to keep track of which item you're on (we'll discuss loops in **Chapter 23**).

TABLE 22-3 lists the available assignment operators in JavaScript.

TABLE 22-3. Assignment operators in JavaScript

Operator	Function	Example	Same as
`=`	Assignment	`a = b`	`a = b`
`+=`	Addition and assignment	`a += b`	`a = a + b`
`-=`	Subtraction and assignment	`a -= b`	`a = a - b`
`*=`	Multiplication and assignment	`a *= b`	`a = a * b`
`/=`	Division and assignment	`a /= b`	`a = a / b`
`%=`	Calculates the remainder after division	`a %= b`	`a` = (the remainder after **a** is divided by **b**)
`**=`	Exponentiation and assignment	`a **= b`	$a = a^b$

For more number and math goodness, I recommend reading the "Basic Math in JavaScript—Numbers and Operators" tutorial on MDN Web Docs (*oreil.ly/U193I*).

For now, though, give numbers and comparison operators a try in EXERCISE 22-2.

EXERCISE 22-2. Paint by numbers

In this exercise, we'll build on the background color changing script by combining it with a conditional statement that applies a different background color based on the width of the browser window.

1. Open the document *color-changer.html* in your text or code editor, if it isn't already open. (If you'd like a fresh start, use *Ex22-1_finished.html*, provided with the materials for this chapter.)
2. Inside the `script` element, add a conditional (`if`) statement (after the existing statement) that checks to see if the browser window is less than 600 pixels wide:

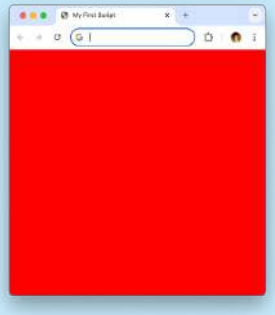
```
<script>
  document.body.style = "background: yellow";
  if ( window.innerWidth < 600 ) {

  }
</script>
```

3. Inside the curly braces of the `if` statement, write another statement that sets the background color to red. This statement will run only if the window is less than 600 pixels wide:

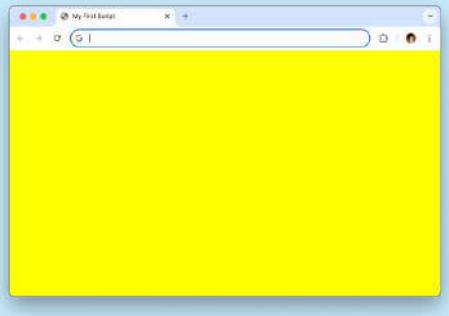
```
<script>
  document.body.style = "background: yellow";
  if ( window.innerWidth < 600 ) {
    document.body.style = "background: red";
  }
</script>
```

4. Save the file and open it in your browser. What background color do you see?
5. If your background color is yellow, make the browser width narrower. If it's red, make it wider. Refresh the page. Now what color do you see? (The results should match FIGURE 22-3.)

Did you notice that the color didn't change automatically when you adjusted the size of the window, as it would have if we'd written this in CSS using media queries? You had to refresh the page to get it to change. That's because the JavaScript we wrote is only run once. We can make it run whenever a user changes the size of their browser window, but to do that we need to know when a user does something. In JavaScript, we access user interactions via events, which we'll discuss in **Chapter 24, Working with the DOM and Events**.



The background is red when the width of the viewport is less than 600 pixels.



The background is yellow when the viewport is 600 pixels or wider.

FIGURE 22-3. The results of using a script to change the background color based on the width of the viewport

STRINGS

In JavaScript, if you see quotes around something, it's a string. Strings are text. They can be letters, numbers, spaces, emojis...pretty much anything. They can also be of any length, from a single character to the entire contents of a web page, code and all. Strings are frequently the best options for storing information, because they can contain anything and everything.

Strings are usually wrapped in quotation marks (or "quotes" in programmer parlance). Both single quotes (`'`) and double quotes (`"`) are allowed (see **Note**). The quote style you choose doesn't matter, but the quote you start with needs to match the one you end the string with (i.e., if you start with a double quote, end with a double quote). Single-quote strings and double-quote strings are equivalent, though you tend to see double quotes used a bit more frequently:

```
// These are the same
"This is my 1st string";
'This is my 1st string';
```

Strings can be any combination of letters, numbers, or symbols. Examples include your name, your mailing address, your email address, and your phone number. As you saw in some of the earlier examples in this chapter, CSS selectors and rules can be JavaScript strings too.

An important rule to know is that a string must be written on one line (see **Note**). If you press Enter while writing a string, the JavaScript engine will get mad at you and refuse to run your code.

NOTE

You can also make strings using backticks (`). These are called **template strings** *(or* **template literals***), and they are a bit more advanced. For an introduction, see the article "How to Use Template Literals in JavaScript" by Benjamin Semah (oreil.ly/Op9SE).*

NOTE

It's OK if a long string automatically wraps onto multiple lines in your code editor or browser console to avoid horizontal scrolling. What the JavaScript engine cares about is hard line breaks with the Enter/Return key. If your coding tool shows line numbers, you'll see that the wrapped lines are not assigned line numbers and are treated as one line of code.

Multiline Strings

Sometimes, the strings you're working with will contain more than one line of text. To make that work without violating the "strings must be written on one line" rule, JavaScript uses the `\n` escape character to represent the shift to a new line (see **Note**):

```
"I'm on line 1. \n And I'm on line 2.";
```

In this example, a newline symbol appears between the two sentences. I added some extra spaces around the `\n` to make it obvious where this symbol is, but they aren't needed. This is a very concise example, but the same technique would be used to preserve line breaks in very long strings (such as large amounts of HTML content that you might insert on a page).

NOTE

You've seen escaped characters in HTML, such as `>` *for the* `>` *symbol. JavaScript also escapes characters that may not be able to be represented in the code. The* `\n` *escape character represents the newline character used to introduce line breaks. For a reference of other JavaScript character escapes, see oreil.ly/egK8A.*

Strings That Contain Quotes

If your string needs to include quoted values (such as attributes in HTML), you have two options: use the other kind of quotes to wrap the internal string, or use the backslash character (\) to escape the quotes. The backslash tells the browser not to treat the character that follows as it normally would. Here's a comparison of the two approaches:

```
'<img src="image.jpg" alt="">';     // Nesting different quote styles
"<img src=\"image.jpg\" alt=\"\">"; // Escaping each quote character
```

If I'm being honest, I find the first one is much easier to read. Sometimes, however, you can't avoid needing the backslash—for instance, when you have a string that contains both single and double quotes 😱😱😱.

Thankfully, that doesn't happen too often!

Concatenation

Earlier, you saw how the **+** operator can be used to add two numbers. You can use that same operator to combine strings too! When the **+** operator is used with text strings, we call it **concatenation**. What do you think will happen if I type the following?

```
"hello" + "world";
```

The strings combine, yes. But did you guess the result would be "helloworld"? That's because neither string included a space. If I want the result to be "hello world," I need to insert the space into one of the two strings or add it as a string between them:

```
"hello " + "world";
"hello" + " world";
"hello" + " " + "world";
```

And yes, as that last example shows, you can concatenate multiple strings in one go.

As I mentioned earlier, strings can contain numbers too. What happens when we combine number strings?

```
"1" + "1"; // "11"
```

In this case, the number characters are pushed together (rather than added), and the result is 11. What do you think will happen when we combine a number with a string?

```
1 + "1";
```

Did you guess the result would also be "11"? If so, well done! If you're confused about why that would happen, I'll explain. Even though the first 1 is a number, JavaScript can't add a number to a string; it can only add numbers

to numbers. Instead of throwing up its figurative hands, JavaScript tries to be helpful and resolve the statement. To do that, it converts the number into a string and, instead of adding, uses concatenation to smoosh the number and the string together. This process of turning one value type into another is called **coercion**, and it happens quite a lot in JavaScript. For a full rundown on how it all works, I recommend reading the GeeksforGeeks article "What Is Type Coercion in JavaScript?" (*oreil.ly/9ofvq*).

Now that you understand a bit more about strings, let's put that knowledge to work in EXERCISE 22-3.

EXERCISE 22-3. Combining strings

In this exercise, we'll build on what we've already done in the previous exercises and make the code a little more dynamic by using a variable combined with the CSS property string. We haven't officially discussed variables yet, but you should be able to follow along.

1. Open the *color-changer.html* file as you left it in EXERCISE 22-2.
2. Replace the code in the `script` element with the following:

```
<script>
  let color = "yellow";
  if ( window.innerWidth < 600 ) {
    color = "red";
  }
  document.body.style = "background: " + color;
</script>
```

   Here we are using the variable `color` to stand in for either the `"yellow"` or `"red"` string (depending on the width of the window), and we're using concatentation to add that string after `"background: "`. The result will be a complete style declaration.

3. Save the file and open it in your browser to make sure it works the same as it did in EXERCISE 22-2 (FIGURE 22-3).

If you copied the code exactly, everything should work exactly as it did before. What we've done here is called a **refactor**. When we refactor our code, we are changing how we've written the code to improve some aspect of it. Sometimes it's an improvement that makes it easier to read. Often, we refactor our code to make it faster. That's what this refactor did.

Every time we touch a part of the document—meaning whenever we reference `document.` *anything*—it slows down the page. With small things it's not very noticeable, but when you start using JavaScript to do a lot in a page, you risk making the page unresponsive. That's why small refactors like this are so important.

This refactor cut the number of times we accessed `document` in half. In the original version, we did it twice: once to make the background red and then again to change it to yellow if the screen was narrow. In this refactor, using a variable name for the color allowed us to access `document` only once. Successful refactor!

By this point, you should be feeling a little more comfortable with numbers and strings. If you'd like to do a deep dive into all things string, including some more advanced topics, MDN offers a great reference at *oreil.ly/tMLTU*.

We'll continue to build on what you've learned so far as you continue through this and the other JavaScript chapters. Now, it's time to switch gears to finally talk about variables.

VARIABLES

In EXERCISE 22-3, I asked you to write the following statement:

```
let color = "yellow";
```

In plain language, this says: I am creating a new variable named "color" and I'm setting its value to the string "yellow". This statement is called a **variable declaration** because it informs the JavaScript engine that there is a new variable to track.

If you're a visual thinker, you can think of variables like buckets (FIGURE 22-4). Buckets are a good analogy, because you can put something in a bucket. In JavaScript, the things we put in these buckets are called **values**. Numbers and strings are two of the many types of values JavaScript supports. You can also label a bucket with some tape and a marker, which is equivalent to naming a variable.

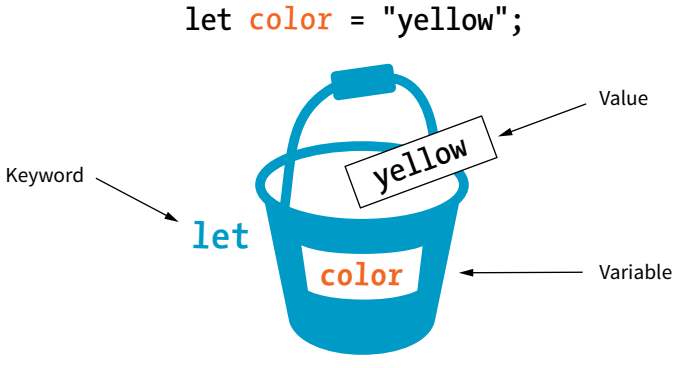


FIGURE 22-4. You can think of variables as labeled buckets that you keep values in to use later.

Variable and function names that I write will appear in orange text.

You can assume any name not in orange is part of the built-in functionality of JavaScript.

In this example, `color` is a variable that I named myself. Going forward, any variable or function name that I create will appear in `orange`, as shown above. You can assume any variable or function name *not* in orange is part of the built-in functionality of JavaScript.

The keyword **`let`** signals to the browser that we're declaring a variable. There are two other keywords—**`const`** and **`var`**—that can also be used to declare a variable. I'll discuss the differences among all three keywords shortly.

The `let` keyword tells the browser to treat the very next thing it encounters as the name of this new variable. With a few restrictions, which we'll discuss in the next section, you can choose any name you want for a variable; however, it's best to be descriptive. In the context of the script we wrote in EXERCISE 22-3, the variable was used for one of two strings—both of which are color names—so "color" was a fitting name for it.

Finally, the equals sign (`=`) tells the JavaScript engine we're assigning a value to the new `color` variable. We call this part of the statement the **initializer** because we're initializing the variable with a value. The value is anything that appears between the equals sign and the semicolon that terminates the statement (e.g., the string "yellow" in the previous example). FIGURE 22-5 shows the structure of a variable declaration.

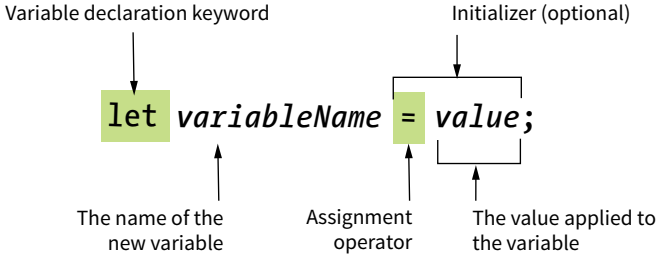


FIGURE 22-5. The structure of a variable declaration

To apply the earlier bucket analogy to the declaration statement that kicked off this section:

1.	`let`	Pull out a new bucket.
2.	`color`	Write "color" on a piece of tape and slap it on the side of the bucket.
3.	`= "yellow"`	Write "yellow" on a scrap of paper and drop it in the bucket.

In the previous example, the value we assigned is a simple string, but it could just as easily be a number or the result of an operation (see **Note**):

```
let grade = 94                          // 94
let minutes_per_day = 60 * 24;          // 1440
let greeting = "hello" + " " + "world"; // "hello world"
```

The first example assigns a number value to the variable `grade` so it can be used elsewhere in the code. In the second example, the number of minutes in an hour (60) will be multiplied by the number of hours in a day (24), and the result of that operation (1,440) will be assigned to the variable `minutes_per_day`. In the third example, the three strings will be combined (a.k.a. concatenated), and the result of that operation ("hello world") will be assigned to a variable named `greeting`.

NOTE

In contexts like these, you may also hear the term "expression" used in place of "operation." There are many kinds of expressions. For more information, check out the MDN article "Expressions and Operators" (oreil.ly/jzFHt).

Naming Variables Is Hard

Coming up with good variable names is one of the most stressful activities you can engage in as a coder. As when applying a **`class`** or **`id`** to an HTML element, aim to be descriptive, but concise. Here are a few rules and pieces of advice that may help:

- You can use nearly any letter character in variable names, though it's best to stick with Latin characters (e.g., `button`).
- Numbers are allowed in variable names, but they cannot be the *first* character (e.g., `button_1`).
- Because variables represent something, it often makes sense to use nouns (e.g., `border`, `total`).
- Good variable names make it immediately apparent what they are referencing (e.g., `cart_total`).
- Parentheses and other operators (e.g., **`+`**, **`-`**, **`=`**, **`>`**, **`<`**) are **reserved characters**, so they can't be used in variable names. You can use nonreserved symbols, such as the underscore (**`_`**) or dollar sign (**`$`**), however.
- Underscores are often used to improve readability (e.g., `submit_button`). We call this naming convention "snake case" because the underscores make it look like the name is slithering along the ground (see **Note**).
- Using a consistent shorthand can help you quickly understand what kind of value a variable holds. For instance, some developers prefix variables referencing HTML elements with a dollar sign (**`$`**), like `$button` (I'll do this in my examples in these chapters). If you're working on a team, you should adopt the team's naming conventions to keep all the code consistent, but if you're working solo you can come up with your own.

NOTE

Because camelCase is used so frequently for function names, I like to use snake_case (all lowercase with words separated by underscores) for variable names as I will be doing throughout this book. It helps me quickly assess whether something is a variable or a function. Many built-in properties in JavaScript also use camelCase, so it's not a perfect approach, but it at least helps with reading my own code.

My approach is, admittedly, a bit of an outlier. As you look at other tutorials and code out in the wild, you will absolutely see variable names written in camelCase. There is no right or wrong; ultimately, it's a matter of preference.

Undefined Variables

It is possible to reserve a variable name without assigning a value to it right away. For example, the declaration:

```
let result;
```

reserves the `result` variable name, assuming you'll assign it a value later. To return to our earlier analogy, we've got the bucket out and we've labeled it, but we haven't put anything inside it. Declaring a variable without assigning a value doesn't happen often, but developers sometimes do this if a variable's value will be the result of a complex series of steps and conditions.

Without an assigned value, the value for this variable is **`undefined`**. This is a special value in JavaScript that is the default value of uninitialized variables.

NOTE

Even though CanIUse.com says over 98% of the current browser market supports `querySelectorAll()`*, that's not 100%. You can use a conditional like this to avoid potential problems in the ~1.5% of browsers that don't support it. That may sound like a small percentage, but depending on the size of the site you're working on, it could actually be a huge number of people.*

While it may seem a little wonky, knowing that **`undefined`** is a thing can be useful. For example, if you wanted to use a new browser feature that isn't widely available yet, you could check to see if it is implemented *before* you use it by seeing if it is **`undefined`**. In this example, I've used the "not equal to" comparison operator (**`!==`**) to make sure that the feature **`querySelectorAll`** is *not* undefined. In other words, I am checking to make sure it *is* defined and, if it is, I can use it safely (see **Note**). It's a little clunky to have the double negative of "not undefined," but there's no straightforward "defined" test, so we make do with what we've got:

```
if ( document.querySelectorAll !== undefined ) {
  // Safe to use querySelectorAll()
}
```

Declaring Variables with let

In EXERCISE 22-3, we declared the `color` variable and assigned it a value of "yellow". Then, in the **`if`** statement, we updated the value of `color` to "red":

```
let color = "yellow";
if ( window.innerWidth < 600 ) {
  color = "red";
}
```

Variables created using **`let`** are called **mutable** because their values can be changed. To go back to the bucket analogy, when you declare a variable with **`let`**, the bucket stays open, and you can replace what's in it with a new value. As this example demonstrates, we can assign a new value to the variable without needing to use the **`let`** keyword again. In fact, you only use **`let`** when declaring a new variable; after it's declared, you use only the variable name when assigning it a new value.

Here's another example, using an assignment operator to reassign a value:

```
let total = 600;
total += 80; // 680
```

This code starts off with `total` being 600. Then it adds 80 to `total` for a new `total` of 680.

We can also declare variables using **`var`** and **`const`**. The **`var`** keyword is an older way of declaring a mutable variable, and it's fallen out of favor (see the sidebar **"The var Keyword"**). The **`const`** keyword is a bit different, in that the variables it creates can't be changed.

Declaring Variables with const

Variables declared using the **`const`** keyword are **immutable**, meaning their values cannot be changed later. When you declare a variable with **`const`**, the bucket is closed tight after you put the first thing in it. The term "const"

The var Keyword

The **`var`** keyword has been around since the beginning of JavaScript. For a long time, it was the only mechanism we had for declaring a variable. In modern JavaScript, its use has been largely replaced by the **`let`** keyword. Newer code generally uses **`let`** and **`const`**, but if you're working on an older project, you may see **`var`** around. You can still use **`var`**, but you're only *required* to use it if you need to support browsers from 2017 or earlier, including Internet Explorer. Those browsers will not run JavaScript that contains **`let`**, **`const`**, and other modern JavaScript because they don't recognize the newer syntax.

Variables declared using the **`var`** keyword have a lot of side effects, whose exploration would take us down a deep, deep rabbit hole. If you like rabbit holes and exploring the arcane and mysterious, you can read up on **`var`** at MDN: ***oreil.ly/jbLWi***. (Heads up: This page is definitely not aimed at beginners.)

is short for "constant." If you try to change the value of a constant, the JavaScript engine will yell at you. I mean, not really, but the browser will put a strongly worded error message in the developer console (we'll discuss the console later in this chapter).

Some developers follow the convention of naming constants in ALL_CAPS to disambiguate them from other variables:

```
const MY_URL = "https://www.aaron-gustafson.com";
```

Now that you know how to declare and use variables, you can take it to the next level in EXERCISE 22-4. When you're done, we'll shift gears a little and talk about where to write JavaScript.

EXERCISE 22-4. Everything counts

In this exercise, we'll explore numbers, strings, and variables a bit more by writing some JavaScript to calculate how long it would take the average person to read a paragraph of text and add this information to the web page. The file, *jens-kitchen.html*, is available with the materials for this chapter.

1. Open the *jens-kitchen.html* file in a text or code editor.
2. First, insert a **script** element into the **body** of the page, just before the closing **</body>** tag (the **script** needs to go at the end so it can reference the HTML above it):

```
  ...
  <script>
    // Your JavaScript will go here
  </script>
</body>
```

3. I like to start by outlining the steps I need to take to achieve a goal right in the **script** by using comments. Add the following comments if you would like a handy road map to follow. You can replace or delete each one as that step is accomplished.

```
<script>
// 1. Find the first paragraph
// 2. Count the number of words in it
// 3. Calculate how long it would take to read
//    (Avg reading speed in US: ~250 words per minute)
// 4. Add the reading time to the end of the paragraph
</script>
```

4. Our first step is to find the first paragraph within the document and assign it to a new variable called **$p** so we can reference it again later in the script. We'll do that with the **querySelector()** method that finds the first element matching the CSS selector string inside its parentheses (in this case, it's a **p** element selector):

```
<script>
  let $p = document.querySelector('p');
  ...
</script>
```

EXERCISE 22-4. Continued

5. Next, we'll add three statements that work together to count the number of words the paragraph contains:

```
<script>
  let $p = document.querySelector('p');

  let text = $p.innerText;
  let list = text.split(" ");
  let words = list.length;
  ...
</script>
```

   Here, I've used three separate operations to get to the information I need. These statements each use a built-in feature of JavaScript (remember that the parts of the code that appear in orange are names I've chosen):

- The **innerText** property of an element is all of its text content (including the text content of its descendants). In the first new statement, we're getting the text content of the paragraph and assigning it to a new variable called `text`.
- The **split()** method breaks text content (a string) into a list of words using a space character (" ") as the divider. We've stored that list of words as the variable `list`.
- Finally, the **length** property, when applied to a list, tells us how many items are in that list. In our example, it gives us the number of words in the paragraph, which we assign to the variable `words`.

6. Now we'll write a statement that calculates approximately how long the paragraph would take to read. We found out that most folks in the US can read 250 words per minute (I made note of it in the comments). Let's create a new variable that represents the reading time (`duration`). We get the `duration` (the number of minutes) by dividing the number of `words` in the paragraph by 250:

```
<script>
  ...
  let duration = words / 250;
</script>
```

7. Finally, we'll update the paragraph to include the reading time at the end:

```
<script>
  ...
  $p.innerText += →
    " (Reading time: " + duration + " minutes)";
</script>
```

   Previously, you saw the **+=** assignment operator used for adding numbers, but it can be used for concatenation as well, to add something to the *end* of something else. Here, we're adding the phrase " (Reading time: *X* minutes)" (where *X* is the value of **duration**) to the end of the paragraph's text content (**$p.innerText**). We get the full phrase by concatenating text strings with the calculated `duration` value.

8. Save the file and open it in your browser. The first paragraph should now include "(Reading time: 0.564 minutes)" at the end (FIGURE 22-6). If you don't see this, double-check that you typed exactly what you see in the code examples, including all punctuation.

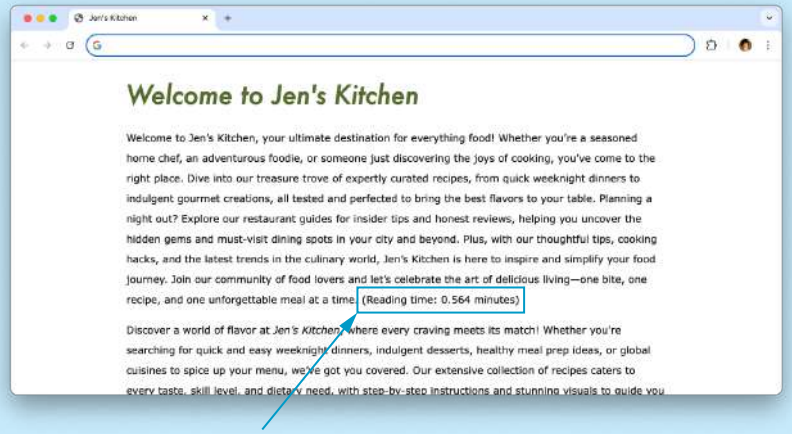

Jen's Kitchen
Welcome to Jen's Kitchen
Welcome to Jen's Kitchen, your ultimate destination for everything food! Whether you're a seasoned home chef, an adventurous foodie, or someone just discovering the joys of cooking, you've come to the right place. Dive into our treasure trove of expertly curated recipes, from quick weeknight dinners to indulgent gourmet creations, all tested and perfected to bring the best flavors to your table. Planning a night out? Explore our restaurant guides for insider tips and honest reviews, helping you uncover the hidden gems and must-visit dining spots in your city and beyond. Plus, with our thoughtful tips, cooking hacks, and the latest trends in the culinary world, Jen's Kitchen is here to inspire and simplify your food journey. Join our community of food lovers and let's celebrate the art of delicious living—one bite, one recipe, and one unforgettable meal at a time. (Reading time: 0.564 minutes)
Discover a world of flavor at Jen's Kitchen, where every craving meets its match! Whether you're searching for quick and easy weeknight dinners, indulgent desserts, healthy meal prep ideas, or global cuisines to spice up your menu, we've got you covered. Our extensive collection of recipes caters to
Our script generated this text and added it to the end of the first paragraph (p).

FIGURE 22-6. The first paragraph now ends with "(Reading time: 0.564 minutes)".

EMBEDDED VERSUS EXTERNAL JAVASCRIPT

Each of the examples and exercises we've looked at so far has been written inside a **`script`** element. We call these **embedded scripts**. They're the JavaScript equivalent of writing CSS in a **`style`** element. It's a useful approach when you're learning or when you're experimenting with new JavaScript functionality. It's also the best approach for code you only need to run on one page (as opposed to across an entire site). As with CSS, however, embedded scripts are not great a great way to manage JavaScript across many pages.

Embedded script code goes directly in the `script` element.

For external scripts, the `script` element is empty and points to a *.js* file via the `src` attribute.

If you embed your JavaScript, users must download the whole program with every page, which isn't great for performance. Thankfully, browsers support **external scripts** as well. If you move your script to a separate *.js* file, you can use it on as many pages as you want, and browsers will use a cached version of the script when they can. This saves your users bandwidth, just like with external CSS files.

JavaScript files aren't special. They're just text files with a *.js* extension. To turn an embedded script into an external script, you can copy the existing code between the opening and closing **`script`** tags, paste it into a file, and hit Save. Once you have the JavaScript file, you reference it from the **`script`** element using a **`src`** attribute (just as you reference a picture file within the **`img`** element):

```
<script src="my.js"></script>
```

JavaScript files make your code a lot more portable. Not only can you tell multiple pages on your site to run a given file, you can also take that file and copy it over to a new project to use there. In fact, that's how many JavaScript libraries began. We'll talk about JavaScript libraries in **Chapter 25, Next-Level JavaScript**.

It's worth noting that external JavaScript programs run in your pages the same way embedded ones do. References to **`document`**, for instance, refer to the HTML file the script is running in, not the JavaScript file. You don't sacrifice anything by moving your code to an external JavaScript file.

To get some experience moving embedded code to a separate file, check out EXERCISE 22-5. When you're done with the exercise, we'll close out this chapter with a discussion of the JavaScript console, which is a handy tool for testing out and debugging your code.

EXERCISE 22-5. Coding outside the page

In this exercise, we'll move the JavaScript we wrote in EXERCISE 22-4 to a separate file and then load that *.js* file into two different web pages.

1. Open the *jens-kitchen.html* file as you left it in EXERCISE 22-4 in a text or code editor.
2. Create a new text file in the same folder and name it *word-counter.js*.
3. Copy the entire script (everything between the opening and closing **`script`** tags) from *jens-kitchen.html*, paste it into *word-counter.js*, and save the file.
4. Go back to *jens-kitchen.html* and erase the contents of the **`script`** element.
5. Add a **`src`** attribute to that **`script`** element and set its value to "word-counter.js":

```
<script src="word-counter.js"></script>
```

6. Copy the **`script`** element, and save *jens-kitchen.html*.
7. Open *aarons-kitchen.html* and paste the **`script`** element into that page's **`body`**, just before the **`</body>`** tag. Save that file as well.
8. Open each page in your browser and verify that the script is running in both pages by looking for the reading time appended to the end of the first paragraph. Congrats! You've written one JavaScript file and are running its code in multiple pages!

If the script isn't running on either page, double-check that your **`script`** element is pointing to the JavaScript file you created. It should be in the same directory as the *.html* files. If you put it in another folder, you may need to update the reference. If it's still not running, be sure the **`script`** element is at the end of the **`body`** element, just before the closing **`</body>`** tag. Remember that if it runs but there is no paragraph before it in the document, it will have no paragraph to reference.

THE JAVASCRIPT CONSOLE

So far, we've been writing scripts in HTML documents and viewing the results in the browser viewport. We can see if it works, but if it doesn't, what went wrong may seem like a mystery. Thankfully, browsers include a tool called the **console** in their developer tools that gives us a glimpse into what the JavaScript engine is doing behind the scenes (FIGURE 22-7).

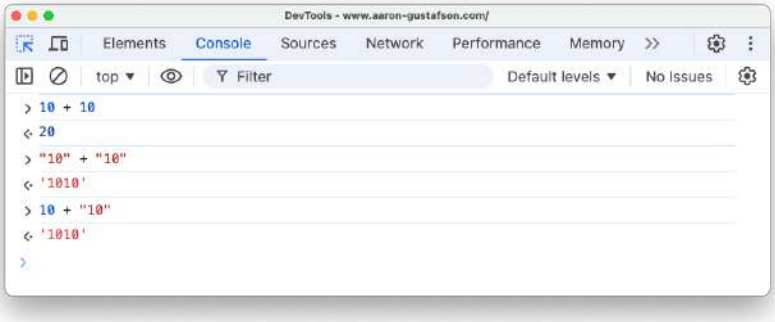


FIGURE 22-7. A screenshot of the JavaScript console in Chrome for macOS. I've typed a few statements into the console to demonstrate how it works.

If you haven't opened the JavaScript console before, look for the Console tab in your browser's developer tools. See the **"Finding the Developer Tools"** sidebar for a refresher on how to access this.

Finding the Developer Tools

All modern browsers make the console available under a Console tab in the developer tools. As a reminder, here's how to access the developer tools in the major browsers:

- **Chrome DevTools: View → Developer → Developer Tools**
- **Safari: Develop → Show JavaScript Console** (if you don't see the Develop menu, check the "Show features for web developers" box in the Advanced tab of the Settings panel)
- **Firefox: Tools → Browser Tools → Web Developer Tools**
- **Microsoft Edge: View → Developer → Developer Tools**
- **Opera: Developer → Developer Tools**

Using the console, you can skip the HTML middleman and type your code directly after the console prompt. The console will show you exactly what your code returns. If it didn't work, it will usually provide you with a helpful error message. This makes the console a terrific tool for JavaScript debugging. It also happens to be an excellent tool for learning and working with JavaScript, so we'll be using it extensively in the demonstrations and exercises going forward. FIGURE 22-8 points out some key parts of the console.

When you open the JavaScript console, you'll typically see an area at the top (the output area) that displays output from any JavaScript code running on the page (including from extensions). You might see some errors, warnings, or other messages, as shown in FIGURE 22-8. If you don't see anything above the prompt, your browser may have collapsed that area until there's content to display in it. Nearby, you'll see a button that clears the console—usually a trash can icon or crossed-through circle (as shown in the top left of FIGURE 22-8). Near the bottom is the place you type your code. It may be a text box, or it could have a prompt indicator like the › shown in the bottom-left corner of FIGURES 22-7 and 22-8.

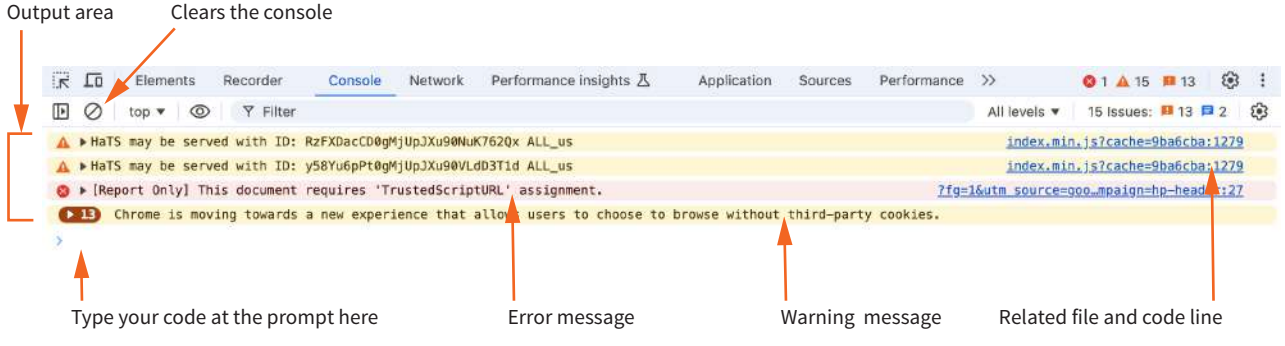


FIGURE 22-8. Key pieces of the JavaScript console (shown in Chrome for macOS)

In addition to using the JavaScript console to test code, you can use it to leave yourself messages with the `console` object's `log()` method. `console.log()` prints anything you put inside the parentheses to the output area. It's a great way to check that your code is producing the expected results (FIGURE 22-9):

```
console.log( "I'm learning JavaScript!" );
```

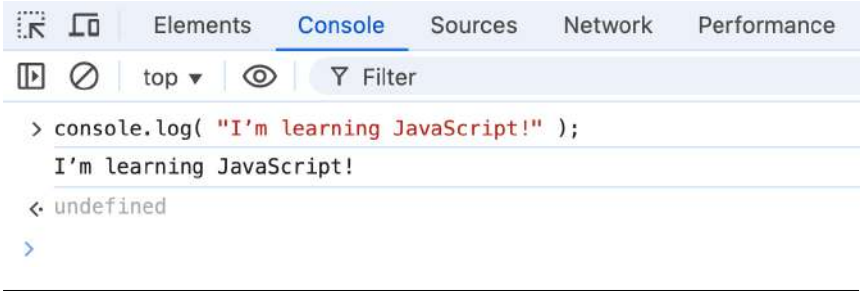


FIGURE 22-9. A demonstration of logging to the console, from the console

Using the Console

The console is a great place to type code and get instant feedback on how it's handled by the browser's JavaScript engine. Once you get your code doing what you want it to do (and error-free!) in the console, you'll need to copy and paste that code into the script element or .js file to save it and apply it to your document.

You can print multiple items at one time as well by separating them with commas. In the following example, I included a string ("The result is") before the current value of `result` to orient myself and provide context. This is helpful if you're calling `console.log()` from several places in your program. I used a comma to separate the parts of the message, but as you can see in FIGURE 22-10, this isn't included in the output:

```
var result = 200 / 10;
console.log( "The result is", result );
```

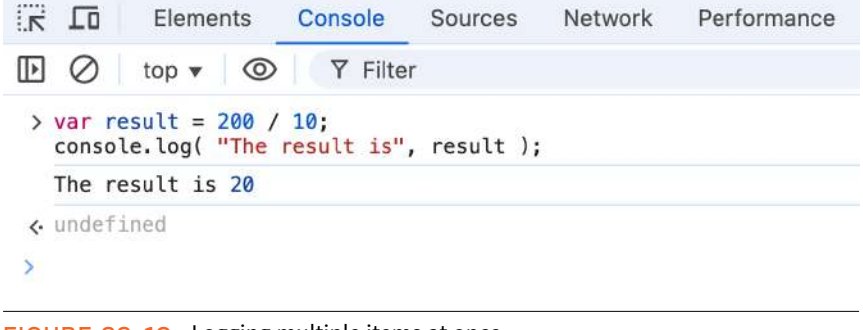


FIGURE 22-10. Logging multiple items at once

You can log to the console at any time, from any JavaScript code, including in external *.js* files.

Get up and running with the console yourself in EXERCISE 22-6.

EXERCISE 22-6. Using the console

In this exercise, you'll type statements into the console and see it run them.

1. Open your browser (the URL of the page doesn't matter), and open your browser's console.
2. Type 1 + 1 after the prompt in the console and press Enter/Return. When you hit Enter/Return, the browser interprets the JavaScript statement you wrote and returns the result (in this case, the number 2) in the output window (FIGURE 22-11).
3. Now type `console.log("Hi from the console")` and press Enter/Return. The console prints out exactly what you told it to inside the parentheses (FIGURE 22-11).

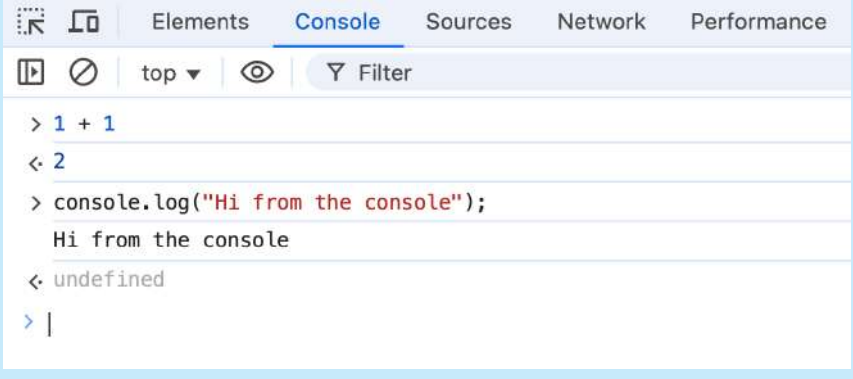


FIGURE 22-11. The resulting console output. Don't worry about the "undefined" text at the end. It only means that `console.log()` isn't returning a value.

Great job! You've used the JavaScript console!

If you want to do a little more, try copying one of the scripts you wrote in a previous exercise (grab everything inside the `script` tags) and pasting it into the console. Then press Enter/Return to run it in the current page.

WRAPPING UP

We've reached the end of the first chapter of your JavaScript adventure. In this chapter, you learned about a whole lot of foundational JavaScript concepts. You learned how to add JavaScript to your page—either directly in your HTML or in a JavaScript file. You learned about number, string, Boolean, and `undefined` value types, and how to assign values to variables. You also learned how to combine and compare values using math operators, assignment operators, comparison operators, and concatenation. Finally, you learned how you can test out JavaScript concepts directly in the browser, using the JavaScript console.

To help you commit the lessons from these pages to memory, I've got two brief challenges for you in EXERCISE 22-7. Neither should take you too much time, but both make use of the syntax you've seen throughout this chapter.

EXERCISE 22-7. Rise to the challenge!

See if you can write the code for the following tasks. This time, you can do your work right in your browser's console and get instant feedback. The answers are provided with the materials for this chapter. There are multiple ways to go about completing these challenges, so your results may not match my final code exactly.

Here are a few tips:

- Start by writing out the series of steps your program should take as JavaScript comments, then fill in the code below each one. This will help you keep track of the steps to ensure you don't miss one.
- If it's helpful, remember you can use `console.log()` to help you keep track of where you are at each step of your program.

Challenge 1: Your age, in seconds

Write math operations to calculate your age, in seconds. We'll do it in a few steps, so you can check your progress in the console along the way. I'll get you started with the first step, but then see if you can get to seconds on your own. Start by declaring a new variable, `days`, whose value is your age (a number) multiplied by the number of days in a year (365.25). Log that result to the console with a hint that it is the number of days:

```
let days = (your age) * 365.25;
console.log( "Days:", days);
```

Next, get the number of `hours` by multiplying `days` by the number of hours in a day (24). Do the same for minutes in an hour (60), and seconds in a minute (60). That's a lot of multiplication!

Challenge 2: Count the paragraphs

Navigate to any page on the web that has a fair amount of text (you can use *learningwebdesign.com*). Write a brief program to count the number of paragraphs (`p` elements) on the page. I'll give you two hints:

1. In the first step, use `document.querySelectorAll()` to get a list of all the elements that match the selector you put inside the parentheses. (Remember, your selector should be a string.)
2. Use the `.length` property to count the result you get from calling `querySelectorAll()`.

TEST YOURSELF

Here are a few questions that check your knowledge of the topics we discussed in this chapter. If you're struggling, you can find the answers in **Appendix A**.

1. Match each variable declaration keyword to its function:

   _____ **`var`**

   _____ **`let`**

   _____ **`const`**

   a. New way of creating a mutable variable

   b. Creates an immutable variable

   c. Old way of creating a mutable variable

2. What are two ways you could write a string containing this HTML? (Pay attention to quote styles.)

   ```
   <a href="https://oreilly.com">O'Reilly</a>
   ```

3. What does **`console.log()`** do?

4. What would happen if you ran this in the console?

   ```
   document.body.innerText = "";
   ```

   a. The content of the **`body`** element would not change.

   b. The **`body`** element would have no content.

5. When you run the following statement, what will the value of `result` be?

   ```
   let a = 4;
   let b = 5;
   let result = a + b;
   ```

   a. 4

   b. 5

   c. 45

   d. 9

6. When you run the following series of statements, what will the value of `result` be? If you need a reminder of what `+=` does, refer to the **"Assignment Operators"** section.

```
let result = 24;
result += 22;
```

   a. 22
   b. 24
   c. 46
   d. 48

7. When you run the following statement, what will the value of `result` be?

```
let result = 12 + "88";
```

   a. 100
   b. "100"
   c. 1288
   d. "1288"

8. When you run the following series of statements, what will the value of `result` be?

```
let a = 12;
let b = "88";
let result = b + a;
```

   a. 100
   b. "100"
   c. "1288"
   d. "8812"

9. When you run the following statement, what will the value of `result` be?

```
let result;
```

   a. `undefined`
   b. Error
   c. `true`
   d. `false`

10. When you run the following code, what will the value of `result` be when it's logged to the console?

```
let result = 1000;
console.log( result += 300 );
```

   a. 0
   b. 1000
   c. 300
   d. 1300

CHAPTER
23

FUNCTIONS, CONDITIONALS, AND LOOPS

by Aaron Gustafson

IN THIS CHAPTER

How to declare and call a function

Passing values to and from functions

Testing for conditions with if and if...else statements

How to program defensively

Creating and using arrays

Making code more efficient with loops

Up until now, all the JavaScript code we've created has been in a series of statements that are run in the order we wrote them. Here is a review of the final code you wrote in EXERCISE 22-4 in the previous chapter that demonstrates this approach perfectly:

```
// Find the first paragraph (p) element in the page
// and assign it to the new variable $p
let $p = document.querySelector( "p" );

// Find the number of words in that paragraph
let words = $p.innerText.split(" ").length;

// Calculate the time required to read that many words
let duration = words / 250

// Add the reading time to the end of the paragraph
$p.innerText += " (Reading time: " + duration + " minutes)";
```

As you'll recall, this finds a paragraph and figures out how long it would take the average adult to read it. Code like this is straightforward and logical. Where it comes up short, however, is that it's not particularly reusable.

If we wanted to calculate the reading time for other paragraphs on the page, we'd have to repeat this code over and over, swapping out the target each time. That's a lot of repetition, and developers hate to repeat themselves. In fact, they even came up with a clever acronym to remind themselves about this: DRY, for Don't Repeat Yourself. We want to keep our code DRY.

This chapter is going to teach you about several JavaScript language features that will reduce repetition in your code and enable you to inject more functionality with fewer lines of code—namely, functions, conditionals, and loops.

FUNCTIONS

As I mentioned when I first introduced them in **Chapter 22, Getting Started with JavaScript**, functions in JavaScript are discrete pieces of code that perform an action. Some functions are part of the JavaScript language itself. Others are a part of the browser. And still others are written by developers like you.

Declaring a Function

To create a new function, you declare it and give it a name with a function declaration statement (in the same way that you declare a variable). There are a few ways to do it, but the most common is this:

```
function doSomething() {
  // Code that does something
}
```

Breaking this down into its discrete parts, the `function` keyword tells the browser you're defining a function. You follow that with the name you want to give it (`doSomething`, in this instance) and a set of parentheses. In this example, the parentheses are empty because the function doesn't require you to pass any information (values) to it when using it. Finally, the code you want to run when the function is called goes inside curly braces. This can be as few or as many statements as you need to accomplish the function's job.

Some functions require that you give them some information. In those cases, when you declare the function, you include one or more value placeholders (called **parameters**) inside the parentheses:

```
function doSomethingWithValues(parameter) {
  //Code that does something with values
}
```

Later, when you call the function, the value that gets passed to the function is called an **argument**. You'll learn more about parameters and arguments in the **"Functions Can Receive and Return Information"** section.

Sometimes, developers declare a function by assigning it to a variable instead. Going back to our bucket analogy from the previous chapter, with this approach you label the bucket with the function name and put a `function()` in it as the value. You'll typically see this when developers want to protect their functions by making them constants. Constants, as you may recall from the last chapter, cannot have their values changed. So, by using the `const` keyword to declare the function, you can create a function that cannot be changed by any other code:

```
const doSomething = function(){
  // Code that does something
};
```

Now that you've seen how to declare a function, let's turn our attention to naming functions.

Naming a Function

Because functions do *something*, they typically have names that make that *something* clear. For example, `querySelector()` looks in (i.e., *queries*) the document for an element that matches a given *selector*. It does exactly what it says. Similarly, `split()` breaks (i.e., *splits*) a string into pieces, based on another string (like the space character used in the earlier example). That's another good name.

When we define custom functions, we should be similarly descriptive. Verbs like *get*, *set*, and *find* are excellent choices for kicking off a function name. From there, you want to be as brief but detailed as possible. Consider these hypothetical function names:

Clear	Less clear
`openNavigation()`	`navigationOpener()`
`makeElementFullscreen()`	`growToFullscreen()`
`getContainingForm()`	`formContainer()`

Most functions are named for what they do, with their names written in camel case (or camelCase), the capitalization convention in which the first letter of every word except the first is capitalized (you may remember this from the first example in the previous chapter). So, "query selector" becomes `querySelector()`, and "get an element by its `id`" becomes `getElementById()`. Although not required, writing function names in camelCase is a practice that's been adopted by most developers.

Calling a Function

To run—or **call**, as developers often say—a function, write its name, including the parentheses (even if they're empty). For example, here is how you would call the `doSomething()` function we declared earlier (see **Note**):

```
doSomething();
```

When the JavaScript engine encounters this function call, it runs the code within the braces of the function declaration. You can call a single function by name as many times as you need to, eliminating the need to write the same code block repeatedly.

Of course, the `doSomething()` function doesn't *actually* do anything yet, because the body of the function is just a comment. In EXERCISE 23-1, you'll get experience writing a function that truly does something.

We used a few functions in the last chapter. Can you remember any of their names? One was `querySelector()`. Another was `split()`. And `log()` was a third. (See the **"Methods"** sidebar for the full story.) If you see some code immediately followed by a set of parentheses, you're typically looking at a function. Exceptions include some built-in statement types like `if`, `while`, and `for` that we'll be discussing later in this chapter.

NOTE

Remember that the function names we create will appear in orange so you can tell them apart from functions built into the JavaScript language. In addition, if I am declaring a function with the `function` *keyword in a code example, you can be sure that the function and its name were created by me.*

Methods

Technically, `querySelector()`, `split()`, and `log()` are called **methods**. Methods are functions that belong to an object; they don't exist on their own. For example, you can't just call the `log()` method; it only works when attached to its object (`console.log()`). We touched on the concept of methods briefly in the **"Objects and Dot Notation"** sidebar in **Chapter 22**.

We've used `document.querySelector()` a few times to find elements in the page, so that one should be familiar. The element that comes before the dot is the **context** in which we're calling the method. So far, we've called the `querySelector()` method on the `document` object (`document` being the whole HTML page), but that's not the only place we can access this method. The `querySelector()` method is available on any element in page. So, for example, if I found a paragraph, assigned it to the variable `$p`, and then called `$p.querySelector("a")`, the `querySelector()` method would only search for `a` elements within `$p`. The element you use before the dot becomes the context for the method. The `querySelector()` method is a key part of the Document Object Model (DOM), which as you'll recall from earlier in the book is how we interact with HTML markup.

You can find the built-in methods available for different value types—such as strings and arrays—by typing a dot (`.`) after the value in the console. To try it out, type `"hello".` in the console; you'll see a list of methods available on strings (FIGURE 23-1). Sites such as MDN (*developer.mozilla.org*) and W3Schools (*w3schools.com*) are good references for learning what each method does and how it works.

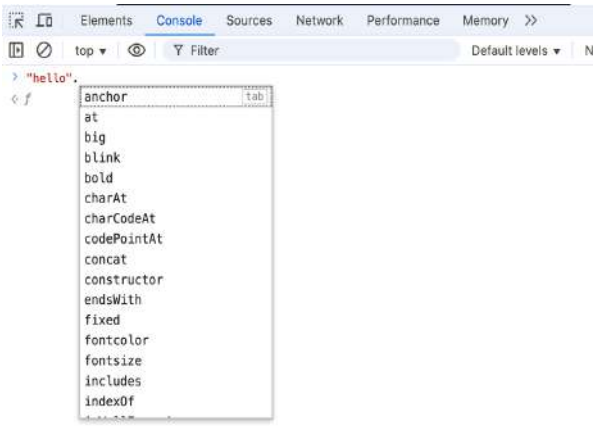


FIGURE 23-1. Typing `"hello".` in the console prompts the browser to suggest properties and methods that are available on strings. Unfortunately, the Chrome browser doesn't include the `()` after method names, but some examples include `charAt()`, `charCodeAt()`, and `indexOf()`.

EXERCISE 23-1. Functional programming

In EXERCISE 22-4, you wrote code for indicating how long it would take to read a paragraph (it was also shown at the beginning of this chapter). In this exercise, we're going to move those statements into a function we can call whenever we need it.

1. Open *word-counter.js* in a text or code editor. Use the file as you left it or get a fresh copy with the materials for this chapter.
2. Declare a new function called `insertReadingTime()` at the top of the file, before any other statements:

   ```
   function insertReadingTime(){

   }
   ```

   This is our function declaration. Now we just need to add the code that will run when we call it.
3. Copy the six statements from the original script and paste them into the body of the function (inside the curly braces). Indenting them as I have here makes it a little easier to read:

   ```
   function insertReadingTime(){
     let $p = document.querySelector( "p" );
     let text = $p.innerText;
     let list = text.split(" ");
     let words = list.length;
     let duration = words / 250;
     $p.innerText += →
       " (Reading time: " + duration + " minutes)";
   }
   ```

   Congratulations! You just created a JavaScript function!
4. At the end of the file, move to a new line and type the following:

   ```
   insertReadingTime();
   ```

   You just called that function!
5. Save *word-counter.js.*, then open *jens_kitchen.html* in a browser (remember that this document references *word-counter.js* in a `script` element). You should see that the code still runs just as it did before, but this time you're using a function. This may seem like a trivial change, but it lays the groundwork for reusing this particular bit of code in later exercises.

FUNCTIONS CAN RECEIVE AND RETURN INFORMATION

In an ideal world, your functions should do one task. The reason is that single-task functions are a lot easier to test and debug when you encounter issues.

If you think about it, the function we created in EXERCISE 23-1 is doing three things:

1. Finding the paragraph
2. Calculating the time required to read it
3. Inserting the result in the reading time calculation and adding it to the end of the paragraph

We can refactor this code to separate out these tasks, but to do that, we first need to discuss how to pass information to and from a function.

Passing Values to a Function

Many functions do something with data (a.k.a. values). In most cases, for a function to work its magic, we need to "pass" it the value it needs inside the parentheses. This should be familiar because we've done it before:

```
document.querySelector( "p" );
```

Here, we've passed the string value **`"p"`** to **`querySelector()`** by putting it inside the parentheses. Now, the function (a method, in this case) knows to look for an element that matches that CSS selector.

Arguments are values that get passed to a function.

When values are passed to a function, we call them **arguments**. When we declare a function, we can define what values we expect to be passed in. We do that by declaring one or more parameters inside the parentheses. Here is the syntax again for declaring a function with an argument:

```
function functionName( parameter ) {
  // Code to execute when the function is called
}
```

The parameter declared in the parentheses corresponds to an argument that we're expecting to be passed in.

Sometimes, a function only needs one value passed in; other times, it requires more values. You can define any number of parameters to a function, but they must be separated by commas, as shown here:

```
function functionName( parameter_1, parameter_2 ) {
  // Code to execute when the function is called
}
```

Enough hypotheticals; let's look at a real example! Here's a simple function that expects two arguments and logs the sum of those values to the console:

```
function add( a, b ) {
  console.log( a + b );
}
```

Again, note the comma separating the parameters in the function declaration. That structure is mirrored when calling the function, at which point you provide the actual data for each argument. Here, the number value 1 is passed into `add()` as the first argument, which is assigned to the parameter `a`. Then, 3 is passed in as the second argument and is assigned to the parameter `b`:

```
add( 1, 3 ); // Logs 4 to the console
```

The parameters `a` and `b` exist only in the context of `add()`; in other words, you can't refer to the parameter `a` elsewhere else in the script and expect it to work. The same holds for variables you declare inside of the function. In JavaScript, we say that both parameters and variables declared within a function are *scoped* to that function. Let's discuss what that means and why it matters.

Variable Scope

When we declare a variable, it exists within a certain context in our application. We call this context its **scope**. Variable scope is an incredibly complex topic, but the most basic scope contexts are **function**, **block**, and **global**:

Function scope

Variables declared within a function exist only in the context of that function. In this example, I'm creating a function called `test()`. Within it, I declare the variable `my_var` and assign it a value of "function scope". Then I log its value to the console. When I call the function, it does all of these things without issue:

```
function test() {
  let my_var = "function scope";
  console.log( my_var );
}
test(); // Logs "function scope" to the console
```

If I changed things up and moved the **`console.log()`** statement outside of the function declaration, when I called `test()` it would fail because the variable `my_var` was declared inside the `test()` function. This means it doesn't exist outside of that function. You will get an error message if you try to reference it outside of the function block:

```
function test() {
  let my_var = "function scope";
}
test();
console.log( my_var ); // Error - my_var is not defined
```

Block scope

Variables declared within a block statement (e.g., an **`if`** statement) are scoped to that block. In the following example, `my_var` is declared within the **`if`** statement block and is recognized only within the curly braces of

that block. It is not available elsewhere in the `test()` function that contains it (see **Note**):

```
function test() {
  if ( true ) {
    let my_var = "block scope";
  }
  console.log( my_var );
}
test(); // Error - my_var is not defined
```

Global scope

Variables declared outside of functions and blocks are referred to as **globally scoped**, which means they are available everywhere. All built-in JavaScript functionality—**`document.querySelector()`**, for example—exists in the global scope, which means we can access those properties and methods from anywhere in our code. This can be a good thing because it means the values or functions are available anywhere we might need them, but the global scope presents challenges too.

Developers generally frown upon putting a lot of code in the global scope because it can create problems later. If you think of the global scope as Earth's orbit, launching new variables and functions into that space makes it more crowded and increases the likelihood of things bumping into one another, to disastrous effect.

For example, if your site makes use of a global variable named `application`, and then a new built-in JavaScript feature comes along and uses that same name, it's highly likely your code will be on the losing end of that collision, which could break your entire site. The same issues can happen in your own code when you have a lot of JavaScript to manage, or when you're mixing code you've gotten from other sources, such as CodePen or GitHub.

To limit how much code is put into the global scope, most developers rely on a practice called **namespacing**. See the sidebar **"Namespacing"** for more on that topic.

Another important concept is scope nesting. While scopes are created by functions and block statements, they nest within one another. A variable's scope begins where it's declared, but it extends to any nested scopes. You can see that in this example, where the global variable `my_var` is available within the scope of the `test()` function and the conditional nested within it:

```
let my_var = "global scope";
function test() {
  if ( true ) {
    console.log( my_var );
  }
  console.log( my_var );
}
test(); // Logs "global scope" twice
```

NOTE

This is not the case if you use **`var`** *(discussed in the sidebar "The var Keyword" in* **Chapter 22***). Variables declared using* **`var`** *are never block scoped. If the code in the block scope example had used* **`var`**, *the* `my_var` *variable would be scoped to the function. This can cause issues with a variable's value getting accidentally changed if the name is reused, which is one reason* **`var`** *has fallen out of favor.*

Further Reading

For more on scope, check out these articles:

- "A Simple Explanation of Scope in JavaScript" by Dmitri Pavlutin (*dmitripavlutin.com/javascript-scope/*)
- "Scope in JavaScript—Global vs Local vs Block Scope Explained" by Adekola Olawale (*oreil.ly/HbBm3*)

Namespacing

As we've discussed, developers try to limit how much of their code sits in the global scope to avoid potential issues with other code running in the browser. At the same time, developers generally want to create code that can be reused beyond the scope of a single program. To resolve that conflict, they will often create **namespaces** in which to collect their code.

To create a namespace, developers attach their functions and variables to a descriptively named object. With the namespace object as the anchor in the dot notation, those functions and variables won't conflict with others in the code with the same names.

Here's an example. I'm going to create a new namespace by assigning an empty object (using an empty set of curly braces) to the variable `MyNamespace`:

```
const MyNamespace = {};
```

Then I can define a new function named `doSomething` and attach it to my `MyNamespace`. Technically it's a method, since it's attached to `MyNamespace`, but we still use the `function` keyword to declare it:

```
MyNamespace.doSomething = function(){
  console.log("hi");
};
```

Note that I didn't need to use `let` or `const` when I defined `doSomething()`, because this function is being attached to `MyNamespace` using dot notation.

I can now access the functionality within `doSomething()` by referencing it as part of `MyNamespace`:

```
MyNamespace.doSomething();
   // Logs "hi" to the console
```

While a little more complicated to set up initially, this approach is more organized and less likely to cause issues later. It's also worth noting that this is just one way to create namespaces in JavaScript.

Some built-in JavaScript functionality is namespaced. For example, the `Math` object provides a namespace for doing all kinds of "mathy" things like rounding and finding the cosine of a number.

For more on namespacing, check out the GeeksforGeeks article "Namespacing in JavaScript" (*geeksforgeeks.org/namespacing-in-javascript/*). If you want an advanced deep dive, read "Essential JavaScript Namespacing Patterns" by Addy Osmani (*addyosmani.com/blog/essential-js-namespacing/*).

Returning a Value from a Function

What makes functions really powerful is that they can also *return* a value that you can use for another task. We've seen this when we supplied a CSS selector to **`querySelector()`** as an argument; that method took the selector, found the element that matched it, and returned a reference to that element so we could access it in our code.

We can also specifically tell a function to stop running and give us a value by using the **`return`** keyword followed by the code that generates the value. For example, I could adjust the previous `add()` function to return the sum of `a + b` rather than logging it to the console:

```
function add( a, b ) {
  return a + b;
}
```

Now, when I call the function, I can capture the returned value and do something with it. In this example, the returned result of the `add()` function is assigned to the newly declared **`sum`** variable:

```
let sum = add( 1, 3 );
console.log( sum ); // Logs 4 to the console
```

Now that you've seen how arguments and return values work, it's time to put this knowledge to use in **EXERCISE 23-2**.

EXERCISE 23-2. Passing values around

In this exercise, we're going to refactor the function we wrote in EXERCISE 23-1 to move the code that gets the number of words from the paragraph into its own function. Then we'll call that function from within `insertReadingTime()`. The finished code is provided with the materials for this chapter if you get stuck.

1. Open *word-counter.js* from EXERCISE 23-1 in a text or code editor.
2. At the top of the file, declare a new function that will count the words and name it `countWords()`. In the parentheses, add a single parameter named `the_text` that represents the text string we'll count words in later. We'll pass that text string in when we call the `countWords()` function in a bit.

   ```
   function countWords( the_text ) {

   }
   ```

3. We want this function to give us (i.e., return) the number of words in a text string (`the_text`), so we'll add a return statement to the body of the function. Return statements begin with the **`return`** keyword, followed by a space and whatever it is you want to return. In this case, we're going to return a chained version of the code we wrote previously. It will split the text string (`the_text`) into words and count them all in one go:

   ```
   function countWords( the_text ) {
     return the_text.split(" ").length;
   }
   ```

   Let's take a closer look at how the chained value works. It starts with the string value (`the_text`) and, using dot notation, calls its **`split()`** method. That method returns a list of words, which is an array value (we'll talk about arrays later in this chapter). Then we find out how many items are in the array using its **`length`** property (again, via dot notation).

   Our `countWords()` function will return the number of words in any text string we pass into it. Let's use it!

4. We can now use the `countWords()` function inside the `insertReadingTime()` function we wrote earlier. The `countWords()` function does the work of the `text` and `list` variable declarations, so those statements can be deleted. Replace the value being assigned to `words` with a call to `countWords()`. Pass `$p.innerText` (which is the text string) as its argument:

   ```
   function insertReadingTime(){
     let $p = document.querySelector( "p" );

     let text = $p.innerText;
     let list = text.split(" ");
     let words = list.length;
     let words = countWords( $p.innerText );
     let duration = words / 250;
     $p.innerText += →
       " (Reading time: " + duration + " minutes)";
   }
   ```

Next, let's repeat the process and extract the words to minutes calculation on the next line into its own function, using the same approach we used for `countWords()`. Declare a new function named `wordsToMinutes()` above `insertReadingTime()` in the same file. This will ensure it's been defined before you use it. It should expect a single argument: the number of words (we'll stick with the name `words`). In the function body, have it **`return`** the result of the words per minute calculation (`words / 250`):

```
function wordsToMinutes( words ) {
  return words / 250;
}
```

5. Now we have a minute-calculating function that we can call from within `insertReadingTime()`. Replace the original calculation with a call to this new function, passing in `words` as the argument:

   ```
   function insertReadingTime(){
     let $p = document.querySelector( "p" );

     let words = countWords( $p.innerText );
     let duration = words / 250;
     let duration = wordsToMinutes( words );
     $p.innerText += →
       " (Reading time: " + duration + " minutes)";
   }
   ```

Great work! You just refactored your code to reduce the responsibilities of the `insertReadingTime()` function. You also practiced sending values to a function and returning them back.

Flexible Functions

Functions are helpful for breaking long or repetitive programs into manageable tasks that are easy to use, test, and debug. Most of the functions you write won't be quite as brief as the ones you worked with in EXERCISE 23-2, but if you find that a function you're writing is doing more than one thing, it might be worth taking a second look to see if you can refactor it into several functions or otherwise change how it operates.

With that in mind, one of the major drawbacks of `insertReadingTime()` is that it only works on the *first* paragraph of a page. What if we wanted the function to operate on *any* (or every) paragraph on the page? Thankfully, we can use what we just learned about parameters to make that happen.

Making this change is a two-step process. First, we need to move the code that finds the paragraph *outside* of the function. The second step will be to pass that paragraph into the function so it can append it with the reading time estimate. Making this simple change will enable us to swap out the one paragraph we've been working with for another paragraph, or even every paragraph.

We'll start by moving the selector code outside of the function (note that this version includes the modifications from EXERCISE 23-2):

```
let $p = document.querySelector( "p" );

function insertReadingTime() {
  let $p = document.querySelector( "p" );
  let words = countWords( $p.innerText );
  let duration = wordsToMinutes( words );
  $p.innerText += " (Reading time: " + duration + " minutes)";
}
insertReadingTime();
```

Interestingly, the code still works as it did before. When we moved `$p` outside of the function, it moved into the global scope, which is still accessible from within the function. But we don't want to access it as a global variable; we want to pass it as an argument to `insertReadingTime()`.

Currently, our function does not expect an argument (the parentheses are empty), so we'll rewrite it to include a parameter in its declaration. We'll keep the name `$p` for the parameter so we don't have to change the body of the function at all. When we call the function, we pass the paragraph element for the parameter (`$p`) directly into `insertReadingTime()`:

```
let $p = document.querySelector( "p" );

function insertReadingTime( $p ) {
  // ...
}
insertReadingTime( $p );
```

Admittedly, this is a little confusing, so let me explain what's going on here. We now have two things named `$p`. One is the variable declared outside of

the function (in the global scope). The other is the parameter scoped to the function. When we pass the globally scoped `$p` variable into the function (as its argument on the last line in this example), the value of that `$p` is assigned to the parameter `$p` within the function.

We don't actually need that outer variable though. We can make the whole thing a little less confusing by directly passing the value returned by the **`querySelector()`** method as the argument for **`insertReadingTime()`**:

```
~~let $p = document.querySelector( "p" );~~

function insertReadingTime( $p ) {
  // ...
}

insertReadingTime(~~ $p~~ document.querySelector( "p" ));
```

With this change in place, we could easily swap the current paragraph reference for any other element, and the code would still work (see **Note**):

```
insertReadingTime( document.querySelector( "body" ) );
```

There is still one problem, however: we don't have any way of ensuring that an argument passed to `insertReadingTime()` is a paragraph, or even an HTML element, for that matter. If we want to protect our function from experiencing an error, we need to do some defensive programming so the function only runs if the argument is an HTML element. But before we can get to that, we need to spend some time talking about conditional statements.

NOTE

If you try this in the console, be aware that resetting the innerText property on the body will wipe away all HTML elements and keep only the text. 😅

CONDITIONAL STATEMENTS

You've seen code samples using the **`if`** keyword a few times in our discussion of JavaScript, but now we're going to take a closer look. The **`if`** keyword is used to create a **conditional statement**. A conditional statement allows you run code based on whether a test condition is met. FIGURE 23-2 shows the syntax for an **`if`** statement (which, by the way, is one of several types of conditional statements in JavaScript).

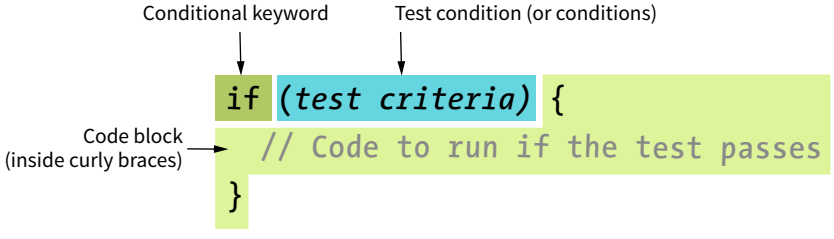


FIGURE 23-2. The structure of an `if` statement

In **Chapter 22**, we used this conditional statement to test whether the viewport was less than 600 pixels wide using the less than (`<`) comparison operator:

```
if ( window.innerWidth < 600 ) {
  // Code for a narrow screen
}
```

You can think of conditionals as asking yes or no questions.

You can think of conditionals as asking yes or no questions. For example, the test condition in this `if` statement asks, "Is the window width less than 600 pixels?" If the answer is "yes" (that is, the result of that comparison is `true`), the test condition passes and the code inside the body of the statement—between the curly braces—is run. If the answer is "no" (the test condition results in a `false` value), the body of the statement will be skipped.

FIGURE 23-3 diagrams how this works for our example `if` statement.

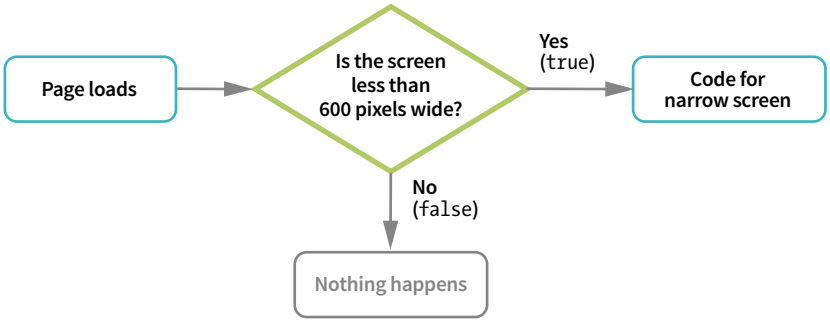


FIGURE 23-3. A logic diagram showing how the `if` conditional statement works

The comparison in a conditional statement can only resolve to one of two values: `true` or `false`. That's it. (It's "yes" or a "no," not a "kinda" or "maybe.") `true` and `false` are known as **Boolean** values. They're typically used in the context of conditional statements like `if` to determine whether certain pieces of code are run or skipped.

If you imagine the curly braces being a fence that restricts access to running the code within them, the condition, with its `true` or `false` value, acts as the gate. When the condition is `true`, the gate is open. When it's `false`, the gate is closed. This analogy isn't too far off, either; sometimes you'll hear developers refer to conditional statements as **logic gates**.

Alternate Code Paths (if...else)

Sometimes, an `if` statement is all you need to accomplish the goal of your script. Often, however, you'll want your code to do one thing if the condition is met and another if it's not. For those instances, you can expand your `if` to `if...else`.

`if...else` statements start with a standard `if` statement block, followed immediately by an `else` statement block. The body of the `else` statement runs only if the answer to the test condition is not "yes" (in other words, it

does not resolve to **true**). FIGURE 23-4 shows the structure of an **if...else** statement. Note that the **else** statement starts immediately after the closing curly brace of the **if** statement.

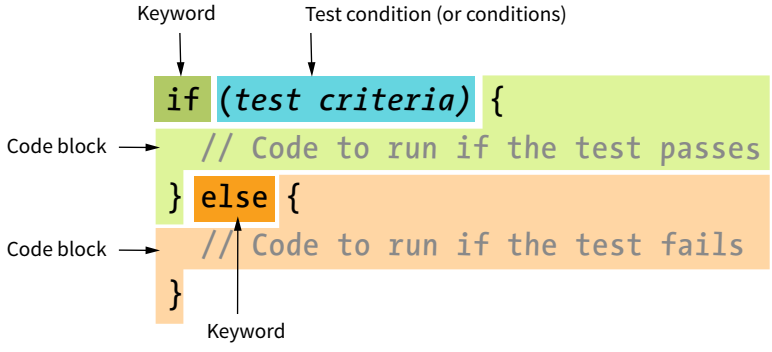


FIGURE 23-4. The structure of an **if...else** statement

Going back to our example, an **if...else** statement allows us to provide a different set of instructions when the window is 600 pixels wide or wider, instead of doing nothing:

```
if ( window.innerWidth < 600 ) {
  // Code for a narrow screen
} else {
  // Code for a wide screen
}
```

FIGURE 23-5 diagrams how it works.

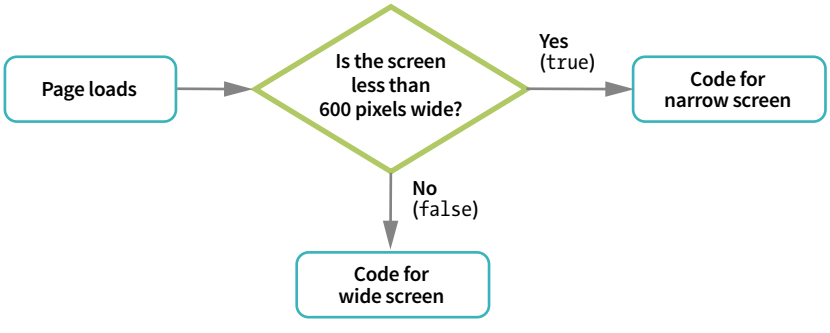


FIGURE 23-5. A logic diagram showing how the **if...else** conditional works

As you gain experience with JavaScript, you'll see **if...else** a lot. There are other variants of the **if** statement as well. For a full rundown, see the **if... else** reference page on MDN Web Docs (*oreil.ly/-mm-T*).

For complex conditions, JavaScript also offers the **switch** statement. To learn what it's all about, read the article "JavaScript switch...case Statement" (*programiz.com/javascript/switch-statement*).

Testing for Multiple Conditions (Logical Operators)

Quite often in programming, we need to run code based on whether multiple conditions are met. This is actually the case with `insertReadingTime()`: we want to ensure that the argument is an element (so `innerText` is available), *and* we want to ensure that the element is a paragraph.

To test for two conditions, we could nest multiple `if` statements, but the better way to do it is to use **logical operators** to connect the test conditions. Let's look at what logical operators are, then circle back.

The logical AND operator (&&)

When you have two conditions that need to be met before your code can run, you can enforce that with the **logical AND operator**, which is two ampersands (`&&`). Here's an example that includes two comparison tests of `window.innerWidth` joined by the logical AND operator. I've added a bit of extra space to clearly show each part, but they could also be written on one line:

```
if (
  window.innerWidth > 320
  &&
  window.innerWidth < 600
) {
  // Code for screens from 321 to 599 pixels
}
```

When the browser hits this point in your program, the body of the `if` statement will be run only if *both* conditions resolve to `true`. So, if `window.innerWidth` were 500, it would pass because 500 is less than 600 but greater than 320. If `window.innerWidth` were 240, however, it would fail because 240 is less than 320. The first comparison statement would resolve to `false`, meaning the whole condition is `false`.

The logical OR operator (||)

JavaScript also supports a **logical OR operator**, which is two vertical pipes (`||`). You use OR when you need only one of your conditions to be `true`. Here's an example that inverts the previous conditional and applies only at screen widths of 319 px or less or 601 px or greater:

```
if (
  window.innerWidth < 320
  ||
  window.innerWidth > 600
) {
  // Code for all screens except those
  // with widths between 320 and 600 pixels
}
```

Testing for the Opposite of a Statement (!)

The final logical operator you'll commonly see is the **logical NOT operator**, which is an exclamation point (`!`). Use NOT to test for the opposite of a statement (i.e., you want to know if the condition is `false` rather than `true`). Here, I'm checking to see if a call to `querySelector()` resulted in a match:

```
let $p = document.querySelector( 'p' );
if ( ! $p ) {
  // Code to run when the page contains no paragraph elements
}
```

If no match is found, the returned value is `null`. `null` is another data type used in JavaScript (add it to the pile with numbers, strings, Booleans, and `undefined`). We say that `null` is **falsy**, which means it's equivalent to `false` when used in a condition, so the code will not run. Check out the **"true/false and Truthy/Falsy"** sidebar for more details on what it means to be falsy.

You can create incredibly complex combinations of logical ANDs, ORs, and NOTs using parentheses to group statements together. Logical operators are a powerful tool.

OK, now that you have a bit of familiarity with conditional statements in JavaScript, let's put this knowledge to use in EXERCISE 23-3.

Further Reading

Take some time to read up on logical operators on the reference page "JavaScript Logical Operators" on the GeeksforGeeks website (*geeksforgeeks.org/javascript-logical-operators/*).

true/false and Truthy/Falsy

As we've discussed, an `if` statement's body will be executed when the test condition(s) resolve to `true`, but not if any of them resolve to `false`. These two values—`true` and `false`—are called Booleans. To reference the example we've been looking at, `window.innerWidth` is either less than 600 or it's not. There's no ambiguity there.

In addition to the rigid `true`/`false` dichotomy, JavaScript also has a concept of `truthy` and `falsy`. These are things that are not explicitly `true` or `false`, but can be interpreted that way. Most things in JavaScript are truthy, so it's easier to focus on which things are considered falsy. Here are a few examples of values that are considered falsy (and therefore will be treated as `false` in a conditional statement):

- `0` (a zero number value)
- `""` (an empty string)
- `null` (an "empty" value, typically meaning nothing was found—in the **"Logical Operators"** section, you saw a `null` value returned when there were no matches to `querySelector()`)
- `undefined` (a variable with no assigned value; we discussed this in **Chapter 22**)

To get an idea of how we can put falsy values to use, let's look at an example. Here, we use the falsy value `undefined` to determine whether a property or method is available:

```
if ( document.querySelectorAll !== undefined ) {
  // Safe to use querySelectorAll()
}
```

This is something you'll see a lot in JavaScript code: testing to see if a property or method—typically a newer feature—is available before you use it. That ensures your code won't encounter an error when trying to call a method that doesn't exist. There's a shorter way to do this, too. Since something that is `undefined` is falsy, something that is defined is truthy. So, instead of writing it as we just did, we can simply write:

```
if ( document.querySelectorAll ) {
  // Safe to use querySelectorAll()
}
```

This says if `document.querySelectorAll` is defined (being defined as a method counts), run the code in the body.

The article "Truthy & Falsy Values and Equality Operator: A JavaScript Guide" by Rabail Zaheer (*oreil.ly/vLY3W*) provides a good introduction to falsy values. You may also consult the Falsy reference page at MDN, which provides a link to the corresponding Truthy resource (*developer.mozilla.org/en-US/docs/Glossary/Falsy*).

EXERCISE 23-3. Making sure the argument is an element

In this exercise, we're going to refactor `insertReadingTime()` to ensure the argument is a paragraph element before proceeding. If it's not, the function won't do anything.

1. Open *word-counter.js* as you left it in EXERCISE 23-2.
2. Before we get to the paragraph test, let's update our function to include the improvements I covered in the **"Flexible Functions"** section. Remove the variable declaration from inside the function. When you call `insertReadingTime()`, pass the result of `querySelector()` directly to the function instead:

```
function insertReadingTime( $p ) {
  let $p = document.querySelector( "p" );
  // Remainder of the function...
}

insertReadingTime( document.querySelector( "p" ) );
```

3. Our first step is to set up the conditional `if` statement that will test whether our text string is an element. This `if` statement is going to contain all of the code from our existing function, so we'll start it at the top of the existing `insertReadingTime()` function, just after the opening curly brace, and end it just before the closing curly brace in the function definition, wrapping the entire contents of the function body:

```
function insertReadingTime( $p ) {
  if ( ) {
    // The function's existing code
  }
}
```

4. The first requirement for our function to work properly is that the argument (`$p`) is an element. We can confirm that by seeing if `$p` has an `innerText` property (which all elements do). If it does, the "Is it an element?" test will resolve to `true`:

```
function insertReadingTime( $p ) {
  if ( $p.innerText ) {
    // The function's existing code
  }
}
```

5. We also need to ensure that `$p` is a paragraph element, specifically. We can use the built-in `.matches()` method to see if a CSS selector for paragraphs would match this element. We use the logical AND operator between the conditions to require that both conditions be met before the rest of the code is run:

```
function insertReadingTime( $p ) {
  if ( $p.innerText && $p.matches("p") ) {
    // The function's existing code
  }
}
```

   Now we're testing for both conditions!

6. Your `insertReadingTime()` function should now look like this:

```
function insertReadingTime( $p ) {
  if ( $p.innerText && $p.matches("p") ) {
    let words = countWords( $p.innerText );
    let duration = wordsToMinutes( words );
    $p.innerText += →
      " (Reading time: " + duration + " minutes)";
  }
}
```

Great work! Now we have a function that only runs the word counter on paragraph (`p`) elements. If we accidentally pass a string, rather than a paragraph element, to `insertReadingTime()`, the code inside the function won't run, and the JavaScript engine won't leave us an angry message in the console.

Defensive Programming

Conditionals are incredibly useful for protecting our programs from running into errors. We often refer to this kind of programming as **defensive**. The code we wrote in EXERCISE 23-3 is an example of defensive programming, but it can show up in other ways as well.

One common way to defend your code is to include a conditional statement at the top of your function that tests for the *opposite* of your requirements (using a logical NOT operator, `!`), then exits the function entirely using `return` if that condition is met. Some developers prefer this, as it keeps the testing piece in one location within the function rather than having it wrap the entire function. Let's see how this approach could be used in the `insertReadingTime()` function.

Here, I've altered the code from EXERCISE 23-3 to test if both conditions are *not* met (using **!**). If the conditions are not met, I leave a helpful message in the console (using **`console.log()`**) that indicates the argument must be a paragraph before exiting the function (using **`return`**). I've broken the whole condition onto multiple lines so you can see the different components and how parentheses are used to group the statements (see **Note**):

```
function insertReadingTime( $p ){
  if (
    ! (
      $p.innerText && $p.matches("p")
    )
  ) {
    console.error( "Argument must be a paragraph" );
    return;
  }

  let words = countWords( $p.innerText );
  let duration = wordsToMinutes( words );
  $p.innerText += " (Reading time: " + duration + " minutes)";
  }
}
```

NOTE

Sometimes you end up with a lot of nested parentheses, which can be confusing. I've added line breaks to help make the nested structure easier to visualize, but you might also see this same **`if`** *conditional statement written on one line, like so:*

```
if (! ($p.innerText && $p.matches("p"))) {
  console.log( "Argument must be a paragraph" );
  return;
}
```

It may seem odd to use **`return`** in this way, since we aren't *returning* a value, but developers commonly use a **`return`** statement like this to stop a function from running based on certain conditions.

With this conditional in place, if I call the function and pass it something other than a **`p`** element, the `insertReadingTime()` function will not run and I see the message in the console:

```
insertReadingTime( "hello there" );
// Console: "Argument must be a paragraph"
```

With the refactoring of `insertReadingTime()` complete, it is now capable of receiving any paragraph as its argument. Developers refer to this as **generalizing** the function because it no longer operates on a specific element. Instead, it works on a category of elements, generally. We've also successfully defended our function from misuse.

Now, let's shift our focus to running this function against every paragraph in the document (not just the first one). To do that, we need a way to find every paragraph element and do something with them. This is a perfect job for arrays and loops, two key concepts in JavaScript that we'll look at next.

ARRAYS

The values we've worked with so far have all been straightforward, single values, like a number or a string. There are more complex value types in JavaScript that can contain multiple values. One of these value types is an **array**.

At a basic level, arrays are lists. No matter who you are or what you do, you probably work with lists all the time. Your to-do list. Your shopping list. The list of books you plan to read once you finish this one.

At a basic level, arrays are lists.

Like lists in real life, JavaScript arrays can be lists of anything: names, prices, product numbers, the elements on a page, the values for a `select` field, every blog post on your site (HTML and all). Literally anything!

JavaScript arrays are represented as a series of comma-separated items inside a set of square brackets (`[]`):

```
[ value_1, value_2, value_3 ]
```

You use array values like any other value type in JavaScript. For instance, here I've declared two new variables and assigned them array values. The first is an array representing a few of my fish, and the second is an array of distances I traveled today:

```
let fish = [ "firefish", "damselfish", "starry blenny" ];
let trips = [ 26.5, 4.1, 15.2 ];
```

As these examples demonstrate, the values in an array can be any valid JavaScript value. The items in an array don't have to be matching value types, either. And array members can even include functions and other arrays!

```
let random = [
  12, "hello", 24, "goodbye",
  function(){
    console.log("what?!")
  }
];
```

Using the Array Index

We reference a particular item in an array by its position in the list, which is called its **index**. Here's the slightly tricky thing: in JavaScript, you start counting items in an array from 0, not 1, as we are all accustomed to counting things in real life. Don't worry, you'll get used to it in no time!

This means the index of the first item in any array is 0, the second item's index is 1, and so on (FIGURE 23-6).

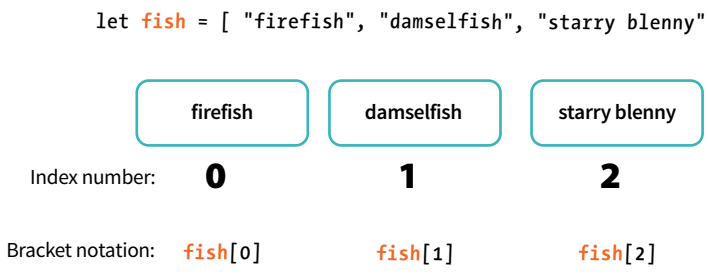


FIGURE 23-6. My array of fish with annotated indices

To access an item in an array, we write the array name followed by the item's index number inside square brackets (referred to as **bracket notation**). If I wanted to access the string `"firefish"` from my `fish` array, for example, I could refer to it in my script using `fish[0]`. The third item in the array has the index 2, so it could be accessed using `fish[2]`:

```
fish[0]; // "firefish"
fish[2]; // "starry blenny"
```

Modifying Arrays

You can make changes to the values inside an array using bracket notation too. To change an existing item, reference it by its index and assign it a new value:

```
fish[0] = "firefish goby";
fish; // [ "firefish goby ", "damselfish", "starry blenny" ]
```

You can also add an item to the end of an array by assigning a value to a new index:

```
fish[3] = "mandarin dragonette"; // Adds a new item
fish;
// ["firefish goby ", "damselfish", "starry blenny", "mandarin dragonette"]
```

There are two other key—for our purposes—aspects of arrays I want to mention. First, we can get the number of items in an array using that array's `length` property:

```
fish.length; // 4
```

Second, we can do something with each item in an array, or list, using *loops*, which we'll discuss in the next section.

Arrays are incredibly powerful, but we don't have time to go too deeply into them here. For an excellent overview of all the cool things you can do with arrays, check out "Understanding Arrays in JavaScript" by Tania Rascia and Lisa Tagliaferri (*oreil.ly/R1sk5*).

LOOPS MAKE REPETITIVE TASKS EASIER

Imagine you have a mailing list of email addresses you want to send the same email to:

```
let mailing_list = [
  "email@domain-1.tld",  "email@domain-2.tld",
  "email@domain-3.tld",  "email@domain-4.tld",
  "email@domain-5.tld",  "email@domain-6.tld",
];
```

Assuming you had written a function to send emails called `sendEmail()`, you could send the emails in a series of statements:

```
sendEmail( mailing_list[0] );
sendEmail( mailing_list[1] );
sendEmail( mailing_list[0] );
// And so on...
```

This isn't very flexible, though, because if you added more email addresses to the list, you'd have to add more statements calling `sendEmail()`. It's also a lot of repetitive code and doesn't respect the developer's mantra of keeping code DRY. Running repetitive code like this for every value in an array is much easier using loops.

Loops run a block of code over and over, a specific number of times. Although they aren't used exclusively with arrays and array-like objects (such as **`NodeList`**, which we'll discuss shortly), loops are most commonly used to run some code against every item in a list. Let's look at a basic loop, and then I'll show you how they can be applied to the `sendEmail()` example.

The for Loop

JavaScript offers several ways to define a loop. We don't have time to get into all of them, so we'll focus on the most common one: the **`for`** loop (see **Note**). Here's an example of a **`for`** loop that counts from 1 to 10 and logs each number to the console:

```
for ( let i = 1; i <= 10; i++ ) {
  console.log( i );
}
```

This may look mysterious at first, so let's explore what each part does.

First, the **`for`** keyword tells the browser this is going to be a loop. The three statements inside the parentheses define how the loop will run, which I will cover momentarily. Finally, the body of the loop—the part inside the curly braces—is the code you want to run in each iteration of the loop.

The three statements inside the parentheses each perform a specific function:

NOTE

Many developers are shifting to using more modern loops, such as **`for...of`** *and* **`forEach()`**, *but some older mobile browsers don't support either syntax. The* **`for`** *loop I've discussed is universally supported and provides a simple way to learn about the functionality of looping.*

`let i = 1;`

Setup. This statement is run only once, before the loop starts. In this example, I used this statement to set the initial value of a variable named `i` to 1 (so the loop will start counting at 1). Developers commonly use `i` to track the current index (hence, "i") of the loop.

`i <= 10;`

Test. This statement, sometimes called the **condition**, is evaluated *before* every iteration of the loop. If the result is `true`, the loop will run; if it's `false`, the loop will stop. Here, the condition is checking to see if the value of `i` is less than or equal to 10. It will stop when `i` becomes 11.

`i++`

Change. This statement runs *after* every iteration of the loop, just before the condition (the second statement) is evaluated for the next iteration. In this example, I'm increasing the value of `i` by 1 by using the incrementing operator (`++`).

When this loop is run, the value of `i` starts at 1. Since 1 is less than 10, it logs 1 to the console. Then `i`'s value is incremented by 1 to 2. As 2 is less than 10, it logs 2 to the console. This continues until `i` is incremented from 10 to 11. At that point, `i` is greater than 10, so the loop ends. FIGURE 23-7 provides a visual representation of the logic here. This example simply logs a number to the console with each loop, but you could use a loop to do more useful tasks for each array item, of course.

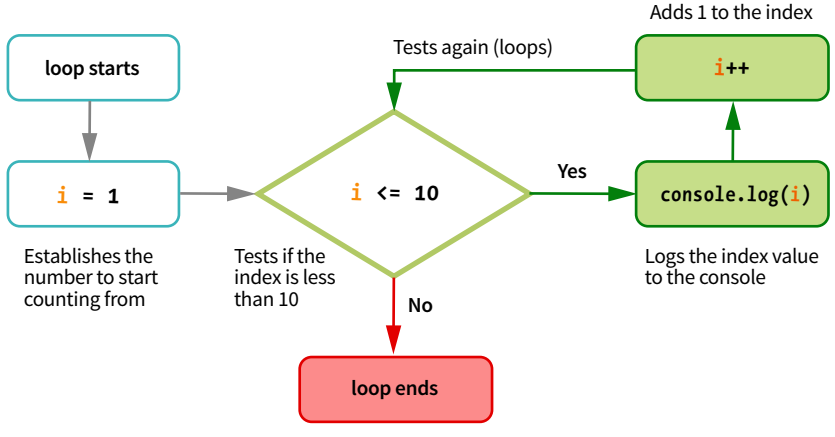


FIGURE 23-7. A visual representation of a loop counting from 1 to 10

Applying what we just saw in the `for` loop to the mailing list example is relatively straightforward. We know the items in an array are indexed using numbers. We also know we can access them through bracket notation. Putting it all together, I created the following loop that calls `sendEmail()` for every email address in the list before stopping:

```
for ( let i = 0; i < mailing_list.length; i++ ) {
  sendEmail( mailing_list[ i ] );
}
```

Here's how I set up the `for` loop:

`let i = 0;`

I'm using the `i` variable to track our position in the array. Since arrays start with an index of 0, `i` needs to start as 0 too.

`i < mailing_list.length;`

We want the loop to run until there are no more email addresses, so we need it to stop when the value of `i` equals the number of items in the array. Every array has a `length` property that provides the total number of items it contains. Conveniently, that number will always be one more than the final index, because the indices start at 0; therefore, we can make a *less than* comparison.

`i++`

Every time the loop runs, we'll increase the value of `i` by 1 so we move to the next item in the array.

Finally, inside the body of the loop, I call `sendEmail()`, passing it the current item in the array.

When the loop initializes, `i` is 0. Because 0 is less than 6 (the `length` of the `mailing_list` array), the loop runs. Inside the body, `mailing_list[0]`, the first email address in the array, is passed to `sendEmail()`. Then the value of `i` is changed, increasing it to 1 for the next iteration. Since 1 is still less than 6, we call `sendEmail()` again, passing it `mailing_list[1]`, the second email address. This continues until we get to the final email address in the list, which has an index of 5. When that iteration of the loop completes, the value of `i` is incremented to 6, which is no longer less than `mailing_list.length`. Since the condition is now `false`, the loop stops.

Loops are incredibly powerful and, alongside functions, are one of the best tools for keeping you from having to write the same code over and over. The `for` loop is one of several ways to loop. Other types include `for...of`, `while`, `do...while`, `forEach()`, and more. The GeeksforGeeks "JavaScript Loops" article (*geeksforgeeks.org/loops-in-javascript/*) provides a good overview. If videos are more your jam, check out codeSTACKr's "JavaScript Loops Made Easy" (*oreil.ly/Rv6vh*).

Now that you know a bit more about arrays and loops, it's time to discuss how all this relates to working with HTML markup.

Iterating over Elements Using Loops

Of the many ways to access the elements in a document, perhaps the simplest is using CSS selectors. You've already done this using `querySelector()`, which returns the first element that matches the selector you provide:

```
let $p = document.querySelector( "p" );
$p; // The first `p` element in the document
```

Sometimes you only need one element, but often you'll find yourself wanting to do something with a collection of elements. To access a collection of elements, we need to use **`querySelector()`**'s cousin, **`querySelectorAll()`**. You can use this method in the same way, but it returns a collection of elements called a **`NodeList`**, rather than a single element:

```
let $paragraphs = document.querySelectorAll( "p" );
$paragraphs; // All `p` elements in the document
```

A **`NodeList`** is not an array, but interacting with one is the same in many ways (which is why we often refer to a **`NodeList`** as an "array-like" list). The elements in a **`NodeList`** each have an index, and the index numbers start with 0. You can also get the number of elements in a **`NodeList`** via its **`length`** property:

```
$paragraphs[0];     // The first `p` element in the document
$paragraphs.length; // The number of elements in the $paragraphs NodeList
```

Are you starting to see how this might apply to our goal of running `insertReadingTime()` on every paragraph element on the page? As we've just seen, a **`NodeList`** has a **`length`** property that tells us how many elements it contains, and we can use bracket notation to access the items within a **`NodeList`**. Combine that with a **`for`** loop like we used for the mailing list, and we're in business!

Meet me in EXERCISE 23-4 and let's put it all together.

EXERCISE 23-4. Loop the loop

In this exercise, we're going to insert the estimated reading time for every paragraph on a page. We'll pick up the *word-counter.js* file where we left off in EXERCISE 23-3 and loop through every paragraph, passing each one to our `insertReadingTime()` function.

1. Open *word-counter.js* from EXERCISE 23-3 (there's also a fresh starter version available with the chapter's materials).
2. Below the function declaration for `insertReadingTime()`, use **`querySelectorAll()`** to gather every paragraph element into a **`NodeList`** (assigned to a new variable, `$paragraphs`) so we can loop through them:

   ```
   function insertReadingTime( $p ) {
     // Existing function code here
   }
   let $paragraphs = document.querySelectorAll( "p" );
   ```

3. Next, capture the number of items in `$paragraphs` and assign it to a new variable named `count`. We'll use this as the upper bound for the loop:

   ```
   let $paragraphs = document.querySelectorAll( "p" );
   let count = $paragraphs.length;
   ```

4. We can now set up the loop. You'll be starting the loop at 0—remember, **`NodeList`**s and arrays are zero-indexed. It should run until you reach the last paragraph, which has an index that is one less than the value of `count`. That means the loop should run while the index (`i`) is less than `count`:

NOTE

You could just as easily reference `$paragraphs.length` *in the loop condition. Your code will run a little faster, however, if you store that value in a variable and use that in the conditional statement instead.*

EXERCISE 23-4. Continued

```
let $paragraphs = document.querySelectorAll( "p" );
let count = $paragraphs.length;
for ( let i = 0; i < count; i++ ) {
  // Code to loop goes here
}
```

5. Now it's time to run our code inside the loop. We want to call `insertReadingTime()` and pass it the current paragraph. The paragraphs are stored in `$paragraphs` and `i` is the index of the current paragraph, so we can use bracket notation to access the current one as `$paragraphs[i]`. Putting it all together, the loop looks like this:

```
let $paragraphs = document.querySelectorAll( "p" );
let count = $paragraphs.length;
for ( let i = 0; i < count; i++ ) {
  // Code to loop goes here
  insertReadingTime( $paragraphs[i] );
}
```

6. Save the JavaScript file and refresh the *jens-kitchen.html* file in your browser. If everything went properly, you should see the reading time added to every paragraph on the page (FIGURE 23-8). Great work!

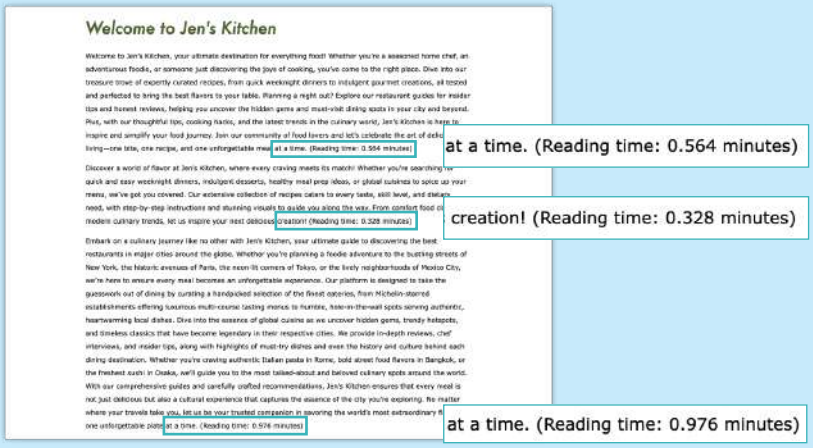

Welcome to Jen's Kitchen
at a time. (Reading time: 0.564 minutes)
creation! (Reading time: 0.328 minutes)
at a time. (Reading time: 0.976 minutes)

FIGURE 23-8. The resulting page with the reading time added to the end of all paragraphs

WRAPPING UP

Let's pause for a moment to reflect on how much you learned in this chapter. First, you learned how to break complex code into bite-sized functions. You got experience passing information to and from functions using arguments and **`return`**, respectively. You saw how conditional **`if`** statements work and learned how you can use them to code defensively. You also learned about arrays, loops, and how to find markup using **`querySelectorAll()`**.

Pat yourself on the back; that was amazing!

Are you up for a new challenge (or two)? I think you are! Head on over to EXERCISE 23-5 and see how far you've come!

EXERCISE 23-5. Challenge yourself!

Now that you have a bit more familiarity with JavaScript programming, try your hand at the following challenges. The solutions are provided with the materials for this chapter.

Challenge 1: Compare element counts

In this challenge, you will use the console to write code that compares the number of **`p`** and **`div`** elements on a web page, then logs a message saying which one there are more of (feel free to use other elements if you'd like). I've provided the logical steps; see if you can convert the steps into statements.

1. Start by navigating to any page on the web and opening the console.
2. Create a variable `div_count` that represents the number of **`div`** elements on the page. (Hint: use **`querySelectorAll()`** to collect all instances into a **`NodeList`**, then use the **`length`** property with dot notation to get the count. Remember that the argument of **`querySelectorAll()`** must be a string (**`""`**).)
3. Create a variable `p_count` that represents the number of **`p`** elements on the page.
4. If the number of **`div`**s is greater than (>) the number of **`p`**s, log "More 'div' elements than 'p' elements" to the console.
5. Otherwise, log "More 'p' elements than 'div' elements" to the console.

Challenge 2: Make a list, check it twice

In this challenge, you'll log a list of things to the console, with a * character inserted before each item in the list. Write the code for each of the following steps.

1. Look around the space you're in, and come up with something you could make a list of. It could be books, devices, writing implements, your fingers and toes…anything. Put that list into an array named `list`. If you need a refresher, look at the `fish` array I created earlier.
2. Next, loop through the items in the array. Each pass will add the asterisk (*) and a space before the item's name. (Hint: follow the example of the loop used in the `sendEmail()` example earlier in this chapter. For each list item (`list[i]`), you'll assign it a new value that includes the string `"* "` using concatenation.)
3. After the loop, use **`console.log()`** to print the array to the console.

TEST YOURSELF

Here are some questions to test how well you retained the information in this chapter.

1. Match the logical operator to its meaning:

   ___ **&&**

   ___ **||**

   ___ **!**

   a. NOT

   b. AND

   c. OR

2. Given the following function definition, what are `a` and `b`?

```
function doSomething( a, b ) {
  // Does something with a & b
}
```

   a. Pass-throughs

   b. Assignments

   c. Declarations

   d. Arguments

3. What keyword do you use to make a function give back a value?

   a. `return`

   b. `respond`

   c. `resolve`

   d. `recall`

4. If you were declaring a function to change the CSS theme being applied to a site, which of the following would be the best name for that function?

   a. `themeChanger()`

   b. `changeTheme()`

   c. `swapTheCSSForThisWebpage()`

   d. `change()`

5. A **`for`** loop contains three statements that fulfill the following roles. Put them in order:

   a. Change ___

   b. Setup ___

   c. Condition ___

6. Which of the following is *not* an example of defensive programming?
   a. Checking that an argument is the value type you expect
   b. Testing to see whether a method you want to use is available
   c. Checking to make sure elements required for your script are available in the page
   d. Only allowing the function to be called between noon and 2 p.m.

7. What is the value of `i` when it is logged to the console?

```
let i = 0;
i++;
i++;
console.log( i );
```

   a. 3
   b. 1
   c. 2
   d. 4

8. How do you access an item within an array?
   a. Square notation
   b. Bracket notation
   c. Brace notation
   d. Bubble notation

9. What will the value of `i` be when this loop stops?

```
for ( let i = 0; i <= 10; i++ ){
  console.log( i );
}
```

   a. 9
   b. 10
   c. 11

10. In the following statement, what is the value type of `$anchors` if the page contains at least one link?

```
let $anchors = document.querySelectorAll( "a" );
```

   a. Array
   b. `ElementList`
   c. `DOMCollection`
   d. `NodeList`

CHAPTER

24

WORKING WITH THE DOM AND EVENTS

by Aaron Gustafson

IN THIS CHAPTER

Progressive enhancement for JavaScript

An introduction to the DOM

Working with element, text, and attribute nodes

JavaScript events

Ensuring accessibility

Over the course of the last two chapters, you learned about a bunch of fundamental JavaScript concepts that serve as the foundation for working with this programming language. This chapter builds on that foundation to give you the skills to do what web developers—like you—are tasked with every day: enabling rich interactions with our web pages.

As we progress through this chapter, we'll be looking at a handful of JavaScript techniques that relate to creating interactive user interfaces, including:

- How to generate new markup and change existing HTML by using JavaScript with the DOM
- How to use JavaScript events that respond to user interactions
- How to ensure your code provides accessible experiences

See the sidebar **"The Plan: Build a Theme Toggle"** for how this will all come together.

I'll kick off the chapter with a refresher on progressive enhancement, a programming philosophy you met back in **Chapter 3, Some Big Concepts You Need to Know**, that will help you make better choices when it comes to designing and building JavaScript-driven user experiences.

This will be a gentle introduction to a more challenging chapter than the previous two. I'm going to be throwing a lot of new syntax and concepts at you, but don't worry if you don't absorb it all right away; concentrate more on *how* we approach the project in this chapter rather than the specific syntax I ask you to write. Mastery of the syntax will come in time, and until then you can always refer to documentation or search the web to find what you're looking for—every developer does. You've got this!

The Plan: Build a Theme Toggle

You'll use the new concepts in this chapter to create a **theme toggle** that enables users to switch between dark and light themes with a button (FIGURE 24-1). Over the course of a few exercises, you'll add a `button` element to the page with JavaScript, make it respond when users click it, and make sure that it's not broken if JavaScript doesn't run.

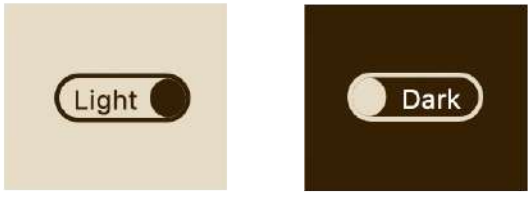


FIGURE 24-1. Light/dark theme toggle button

UI AS A PROGRESSIVE ENHANCEMENT

The theme toggle we'll create in the exercises is going to be a purely JavaScript-driven interface. That means it won't work if JavaScript isn't available or working as expected. Yep, that's a possibility.

It doesn't happen often, but sometimes the JavaScript we write won't run (see **Note**). Sometimes it's because the user loses their internet connection while trying to download the resources for our site. Other times, a browser plug-in might create an issue by blocking a script or making changes to the DOM that the script wasn't expecting. Or maybe the JavaScript we wrote uses a feature the user's browser doesn't support.

NOTE

For an overview of many of the reasons your JavaScript may not run, check out "Everyone Has JavaScript, Right?" by Stuart Langridge (oreil.ly/GQeY1).

This underscores the importance of writing JavaScript programs that don't cause problems when they can't run. This is a central tenet of **progressive enhancement**, which, as you learned back in **Chapter 3**, is an approach that encourages us to build a baseline experience that works for everyone, and then use markup, CSS, and JavaScript to enhance their experiences when we can.

For example, you can use progressive enhancement to provide alternatives for how clicking a thumbnail image takes the user to the larger version of the image. The default experience may be to load the image in its own window (or a new tab), which can be handled with HTML alone, while the progressively enhanced experience could be to use JavaScript to load the larger image in a "lightbox" overlaying the current web page. Another example is a form that saves what you type as you fill it out, in case you accidentally close the

tab before submitting it. The form still works without JavaScript, but if you do have it and the worst happens, you haven't lost all your work.

How does progressive enhancement apply to our theme toggle? We're not going to put any markup for the UI in the HTML file. Instead, we're going to create the markup and insert it into the page using JavaScript. Approaching this project that way ensures that users without JavaScript won't see a button that doesn't do anything. If the JavaScript doesn't run, they won't even know that there's a theme toggle button to begin with. But if it does run, the theme toggle will appear and work perfectly (see FIGURE 24-2).

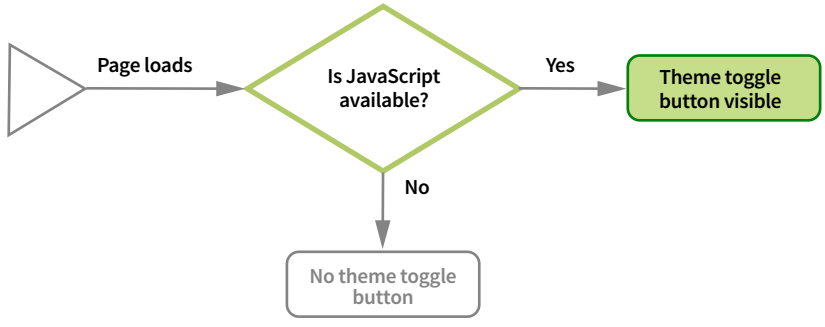


FIGURE 24-2. An Interface Experience Map (Ix Map) defining the potential experiences, with the difference being whether our JavaScript code is run. For more on Ix Maps, see the article "Interface Experience Maps" by me, Aaron Gustafson (*aaron-gustafson.com/notebook/interface-experience-maps/*).

Progressive enhancement is often discussed in relation to JavaScript, but as mentioned previously, it can also be implemented with HTML or CSS. The `download` attribute on a link will trigger modern browsers to download the linked resource, whereas older browsers will just follow the link to its destination. Similarly, the `autocomplete` attribute enables you to tell a browser how to fill in certain form fields automatically, if they support the feature.

In all these cases, the default experience is usable. Then, we enable a better experience where and when we can.

GENERATING NEW MARKUP

Throughout **Part II, HTML for Structure**, you wrote a lot of HTML documents and loaded them in the browser. In those documents, the content in the HTML source is what renders on the screen, and it's **static** (that is, it stays the same for all users, always). But I'm sure you've seen web pages with content that changes on the fly. A shopping site might display more product listings when you scroll. You might click a "Show More" button to load more content. A site may greet you with "Good Morning!" in the morning and "Good Evening!" later in the day.

FURTHER READING

For a deep dive into progressive enhancement, I recommend reading my book, *Adaptive Web Design*. The first edition is available for free at *adaptivewebdesign.info*, and you can get the second edition at your favorite bookstore.

This is a job for JavaScript! It's common to use JavaScript to make changes to a document's markup and content dynamically. Those changes might be based on user actions, such as clicking or scrolling, but you could also change up content based on other criteria, such as the time of day or the device users are viewing it on.

To access and manipulate a page's markup, including the elements, their attributes, and any content they contain, we use JavaScript with the Document Object Model (DOM).

The Document Object Model

You've seen references to the DOM in several previous chapters, but what is it, really?

You can think of the DOM as a family tree representing everything in the document. As you learned in **Part III, CSS for Presentation**, in CSS we refer to elements as children, descendants, siblings, and parents. Those same relationships are represented in the DOM, but it gets much more granular, because elements aren't the only members of this family tree. We refer to the various components of a document as **nodes**. As well as element nodes, the DOM includes text nodes (representing the text within elements), comment nodes, and attribute nodes. FIGURE 24-3 shows how a single paragraph element node is made up of other nested nodes, all of which can be accessed and manipulated with code.

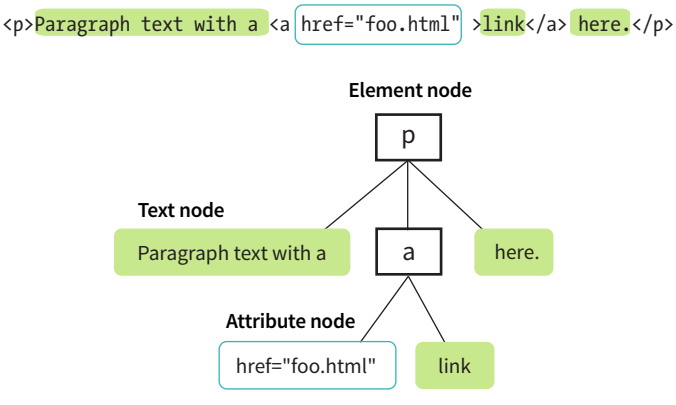


FIGURE 24-3. Breaking a paragraph element (p) into a collection of DOM nodes

■ FURTHER READING

For a good explanation and overview of the DOM, I recommend the MDN Web Docs article "Introduction to the DOM" (*oreil.ly/HVfCb*).

We use the various properties and methods available to us as part of the DOM to move around in this rich and detailed family tree. We can also use them to prune certain nodes, sprout new ones, and adjust others to achieve our goals.

The document Object

Let's begin our exploration with the **document** object, at the root of the DOM tree. You've seen the **document** object in many examples and exercises already, but here you'll get to know it a little better.

The **document** object represents the current HTML document loaded in the browser. It often serves as the starting point for navigating to other nodes on the page.

The **document** object has numerous properties that provide shortcuts to specific element nodes, such as **body** (the **body** element of the document), **title** (the page's title), **links** (a collection of **a** elements in the document), and **documentElement** (the **html** element). We access these using dot notation (e.g., **document.body**).

The **document** object also has a number of built-in methods. You've used a few of these already, such as **querySelector()** and **querySelectorAll()**, and you'll meet several more in this chapter.

For a complete list of the properties and methods of the **document** object, see the reference page on MDN (*developer.mozilla.org/docs/Web/API/Document*).

Now that you've got a better understanding of the **document** object, let's use a few of its methods to create new markup!

Creating and Inserting Element Nodes

To create a new element node to inject into the document, we use the DOM method **createElement()**. This method is available on the **document** object only. It takes a single argument, which is the tag name of the element you want to create.

If I want to create a new paragraph, for example, I can call **createElement()** and pass "p" as the argument. Here I've done just that and assigned the new paragraph to a variable, $new_p:

```
let $new_p = document.createElement( "p" );
```

At this point the element is not actually in the markup of the document. Technically, it's sitting in my computer's memory because it hasn't been attached to the current page yet. Even in its detached state, however, you can still work with a newly created element like any other value you drop into a variable "bucket."

To get an element into the page, we have several options, depending on where we want to put it. I'm going to show you the two most common methods: **appendChild()** and **insertBefore()**. And because we're adding elements, it seems only fair to discuss removing them too. We'll look at **removeElement()** before we close out the section.

appendChild()

The **appendChild()** method inserts an element at the end of an existing element's children. In other words, it *appends* the new element as a *child* of the existing element. That's a well-named method 😉.

You start with the existing element, then call its **appendChild()** method using dot notation and pass in the new element as the argument (inside the parentheses). In this example, I am adding the new paragraph I just created (stored as the variable $new_p) to the **body** element:

```
document.body.appendChild( $new_p );
```

This injects the new paragraph element at the end of the **body** element, below any existing children. Now the paragraph we created is part of the document and will display at the bottom of the page when it loads.

insertBefore()

Maybe you don't want the new element to appear as the last child of its parent. In that case, you can use the **insertBefore()** method to specify which element it should appear before. Like **appendChild()**, you call this method on the element you want to be the parent of the new node. The **insertBefore()** method takes two arguments: the first is the new node you're looking to insert, and the second is an existing child of the soon-to-be parent element that you want to *insert* the new element *before*.

So, if I wanted to make $new_p the first element inside the **body** instead of the last—right up at the top—I could do it like this:

```
document.body.insertBefore( $new_p, document.body.firstChild );
```

Here, I'm saying I want to make $new_p a child of the **body** element (**document.body**), but I want it to be inserted before the **body** element's current first child node (**document.body.firstChild**), regardless of what that child happens to be (a heading, a **div**, another paragraph, etc.). In this example, I used the **body** element's first child as the reference element, but I could just as easily have used any of its child nodes, and it would have inserted the new paragraph before that node instead. You'll see an example of how to reference a specific child node in the next section.

removeChild()

Just as you can add nodes, you can remove them too. As the name suggests, the **removeChild()** method takes an existing element out of the DOM. Like the previous two methods, you call **removeChild()** on the parent element (thus, you *remove* a *child*). It takes one argument, which is the element you want to remove.

In this example, I'll remove the second **h1** element from the page. I'm going start by getting the element (using an nth-child pseudo-class selector) and assigning it to a variable:

```
let $second_h1 = document.querySelector( "h1:nth-child(2)" );
```

I can then remove it from its parent using the element's **parentNode** property (another part of the DOM):

```
$second_h1.parentNode.removeChild( $second_h1 );
```

The **parentNode** property will always refer to the reference node's parent. It could be the **body** element, a **div**, a **section**, or something else entirely.

> **AT A GLANCE**
>
> Here's an overview of the syntax for the methods covered in this section:
>
> *parent*.appendChild(*new node*)
>
> *parent*.insertBefore(*new node*, *existing element it goes before*)
>
> *parent*.removeChild(*existing node to remove*)

Working with the Element Insertion and Removal Methods

Let's look at a longer example that employs **appendChild()** to insert a paragraph into a page and **insertBefore()** to add a new list before it. If you'd like to follow along, you can open the JavaScript console on any web page and type these commands.

First, I'll get a reference to the **body** of the page:

```
let $body = document.body;
```

Then I'll create a new paragraph:

```
let $p = document.createElement( "p" );
```

Remember, it's not in the page yet. Right now, this paragraph only exists as the variable $p. For it to show up, I need to add it to the page, which I'll do using **appendChild()**. Before I do that, however, I should add some content to it by assigning a string value to its **innerText** property:

```
$p.innerText = "I made this with JavaScript.";
$body.appendChild( $p );
```

Once appended to the **body**, the new paragraph shows up at the bottom of the page, as shown in FIGURE 24-4. (If you're following along, you may need to scroll down to see it.)

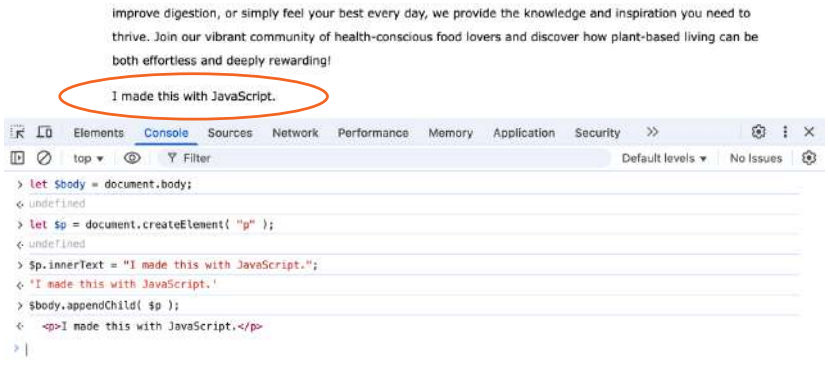

improve digestion, or simply feel your best every day, we provide the knowledge and inspiration you need to thrive. Join our vibrant community of health-conscious food lovers and discover how plant-based living can be both effortless and deeply rewarding!
I made this with JavaScript.
Elements Console Sources Network Performance Memory Application Security
top Filter Default levels No Issues
> let $body = document.body;
undefined
> let $p = document.createElement("p");
undefined
> $p.innerText = "I made this with JavaScript.";
'I made this with JavaScript.'
> $body.appendChild($p);
I made this with JavaScript.

FIGURE 24-4. Adding a paragraph to the bottom of a page using `appendChild()`. (I used the *aarons-kitchen.html* file, but you can do this with any web page.)

Now I'm going to use **`insertBefore()`** to insert a new unordered list (**`ul`**) before that paragraph. First, I'll create the list (`$ul`) and a single list item (`$li`) and give that item some content:

```
let $ul = document.createElement( "ul" );
let $li = document.createElement( "li" );
$li.innerText = "I made this with JavaScript too!";
```

If I added the list to the page at this point, I wouldn't see it because it doesn't contain any list items. I created one (`$li`), but I need to attach it to the list. Time to call **`appendChild()`** again, but with `$ul` in the parent slot:

```
$ul.appendChild( $li );
```

Now the list is ready to be added to the page. As I mentioned, I want to add it to the **`body`** element, but I want it to come *before* the paragraph I added. To do that, I'll use **`insertBefore()`**. The `$body` variable will be the parent element, and I can use `$p` as the reference element to insert it before:

```
$body.insertBefore( $ul, $p );
```

FIGURE 24-5 shows the result.

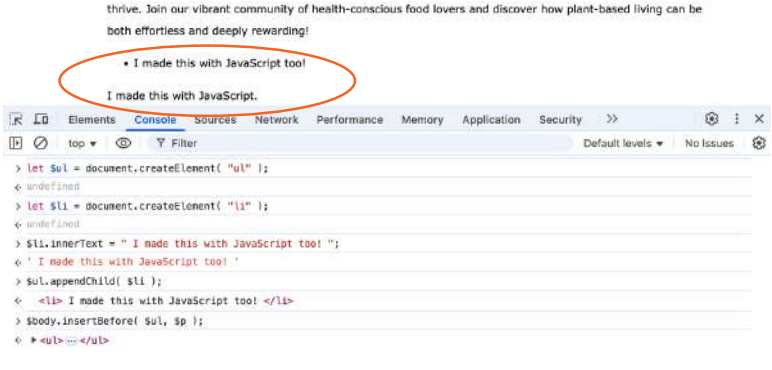

thrive. Join our vibrant community of health-conscious food lovers and discover how plant-based living can be both effortless and deeply rewarding!
• I made this with JavaScript too!
I made this with JavaScript.
Elements Console Sources Network Performance Memory Application Security
top Filter Default levels No Issues
> let $ul = document.createElement("ul");
undefined
> let $li = document.createElement("li");
undefined
> $li.innerText = " I made this with JavaScript too! ";
' I made this with JavaScript too! '
> $ul.appendChild($li);
 I made this with JavaScript too!
> $body.insertBefore($ul, $p);
▶ …

FIGURE 24-5. Adding a list with one item before the new paragraph (`$p`) using `insertBefore()`

Now that we've seen how to insert and delete elements individually, let's look at a way to make large-scale adjustments to markup using **innerHTML**.

Adding Large Amounts of Text

I've kept examples in this chapter short and sweet by adding only small phrases of text with the **innerText** property. Of course, it's possible to use **innerText** for longer content as well. And if you want to include markup in the text string, we also have the **innerHTML** property. This section looks at both options for adding text content to elements.

The innerHTML property

We've been sticking with the **innerText** property for adding text content to an element, but that's not the only option. If you want to capture or insert content *including the markup*, use **innerHTML**. This property is available for all elements in the DOM, and it represents the content you see when you view a page's source (as described back in EXERCISE 2-1). This would be the tool to use to access or add content like a product listing or a whole article without losing important semantic information and structure.

To give a simple example, I could use **innerHTML** to grab all of the content on a page. If I were to type the following statement in the console, it would resolve to a (potentially long) string that contains all the HTML content of the **body** element:

```
document.body.innerHTML; // all HTML content of the body element
```

Similarly, I can use **innerHTML** to replace the entire contents of an element with new HTML, like I'm doing here with the **body**:

```
document.body.innerHTML = "<p>Poof!</p>";
```

If I were to run this statement in the console, the entire content of the **body** element would be replaced with a single paragraph containing the text "Poof!"

As you can see, the **innerHTML** property is incredibly powerful. I often liken it to a sledgehammer, in that if you're not careful how you wield it, you're probably gonna break something. The DOM methods we just looked at—**createElement()**, **appendChild()**, etc.—are more like scalpels, allowing you to work with surgical precision.

That's not to say **innerHTML** isn't worth your time, though. There will be instances where it's more efficient to write a whole block of HTML in one go rather than one piece at a time. For example, you might use **innerHTML** to inject a summary of errors that appear in a form a user is filling out; doing this in one go is much faster than writing each error as a list item, attaching that list item to the list, and then inserting the list into the page.

If **innerHTML** feels familiar, that's because it's closely related to **innerText**, which we've used previously. The only real difference is that **innerHTML** includes any HTML markup.

Now that we have a good sense of the different ways to insert markup, let's turn our attention to text content.

The innerText property

We've been using **innerText** for several chapters now, but let's give it a closer look.

The **innerText** property represents the text content of an element as it is rendered, including all of its descendants—that is, the text you see in the browser when the page loads. Essentially, it's **innerHTML** without the HTML. You don't see markup in a browser, and you also don't see any text that is "hidden" with CSS (see **Note**).

NOTE

A third text property, **textContent**, *will access hidden text as well. I won't be covering it here, but you can find information about it in the reference I provide at the end of this section.*

Consider this HTML navigation as an example:

```
<nav>
  <p>Here's what you can find on this site:</p>
  <ul>
    <li><a href="/about/">About</a></li>
    <li><a href="/notebook/">
      <b class="hidden">Entries in My </b>Notebook
    </a></li>
  </ul>
</nav>
```

Here, I'm getting a reference to the **nav** element and then logging its **innerText** to the console:

```
let $nav = document.querySelector( "nav" );
console.log( $nav.innerText );
```

What I see in the console is this:

```
Here's what you can find on this site:

About
Notebook
```

There are a few things to notice in the resulting string. First, the HTML is stripped out, as expected. Second, the line breaks are preserved. Finally, the **b** element, which has its **display** set to **none** (via the **class** "hidden"), is not included because it is not visible in the browser window.

innerText is a fine option for adding text to new elements—as we've done in some of the examples—but, like **innerHTML**, it's a bit of a sledgehammer. It's great for writing big chunks of text, but a big limitation is that it doesn't consider HTML at all. In fact, **innerText** makes it a little too easy to accidentally strip all the markup out of a particular element and leave only the unformatted text behind if you're not careful.

■ FURTHER READING

To learn more, read the article "innerHTML vs innerText vs textContent—What's the Difference?" by Benjamin Semah (*oreil.ly/_Wcdb*).

For more control over text content, it's best to stick with the surgical DOM methods we'll discuss in the next section.

Creating and Inserting Text Nodes

Thus far in our journey, we've reached for the **innerText** property when we needed to add content, but as we just discussed, that approach is not always the best option. In this section, we'll look at the **createTextNode()** method as a more precise means of inserting text into a page. We'll also see how **nodeValue** can be used to change the text in a text node.

The createTextNode() method

Much like we can use **createElement()** to generate a new HTML element, we can use **createTextNode()** to create new text content. In case you're wondering, it's called "createTextNode" instead of just "createText" to reinforce that text blocks exist as nodes within the DOM, just like elements do. Remember, everything in the DOM is a node.

To create a text node, you call this method on the **document** object and pass in a string (your text content) as the argument:

```
let new_text = document.createTextNode( "Hi there!" );
```

Once you've created the node and assigned it to a variable, you can insert it into the page using the same DOM methods we use to insert elements: **appendChild()** and **insertBefore()**. Let's look at an example to make this a little more concrete.

I'm going to insert the new text node I just created ("Hi there!") at the beginning of the paragraph we created earlier in this chapter (**$p**). To do that, I'll use **insertBefore()** with **$p** (the paragraph) as the parent and its first child (the text node "I made this with JavaScript!") as the reference point. I can reference that text via the paragraph's DOM property **firstChild**:

```
$p.insertBefore( new_text, $p.firstChild );
```

FIGURE 24-6 shows the logic behind how it works.

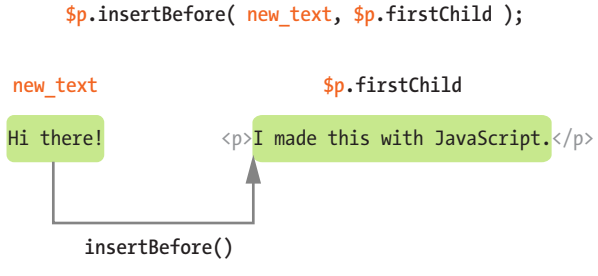


FIGURE 24-6. A visual illustration of inserting one text node (new_text) before another one (**$p.firstChild**) within a paragraph element (**$p**)

FIGURE 24-7 shows the results. You'll notice that the two text nodes in the paragraph run together because we didn't include a space. If you recall, we ran into a similar issue when concatenating strings in **Chapter 22, Getting Started with JavaScript**.

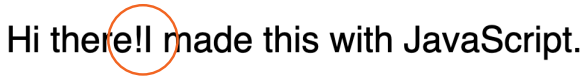


FIGURE 24-7. Oops! The new text node we inserted ran into the existing text because we forgot to include a space.

The nodeValue property

No worries, we can fix that! Sure, we could have included the space when we declared `new_text`, but we can also "fix it in post," as they say. Doing that gives us a chance to look at one more piece of the DOM that can be useful: **`nodeValue`**. The **`nodeValue`** property allows you to update the string value (the text content) of a text node, just like you'd assign a new string value to a variable. It works like this:

```
text_node.nodeValue = "Some new string value";
```

We can use **`nodeValue`** to update the value of `new_text` to include a character space at the end. We add the space at the end by appending (**`+=`**) a space character (see **Note**):

```
new_text.nodeValue += " ";
```

As FIGURE 24-8 shows, running this code inserts the space at the end of the new text node (`new_text`), which gives some separation between the two sentences.

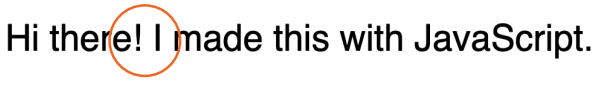


FIGURE 24-8. That's better. There's now a space between the two sentences.

Are you ready to manipulate the DOM? In EXERCISE 24-1, you can get some hands-on experience creating an element and changing its content.

Now that you know how to insert both elements and text content, let's shift our focus to attribute nodes.

NOTE

The assignment operator **`+=`** *adds what comes after the operator to the end of whatever came before it. We discussed it in the context of numbers in* **Chapter 22***, but it can be used for concatenating strings too. In this case, it's using string concatenation to add a character space (" ") at the end of "Hi there!" See* **Chapter 22** *for a refresher on* **`+=`**.

Creating, Updating, and Removing Attribute Nodes

At this point in your web design journey, you should be quite familiar with HTML elements and their attributes. As you learned in **Part II**, some attributes are required, like the **`src`** and **`alt`** attributes on an **`img`**, while others are optional, like the **`id`** attribute, which can be added to any element.

EXERCISE 24-1. Building a Button

In this exercise, you're going to create a **button** element and add it to a page. The *theme-toggle.js* and *toast.html* documents are available with the materials for this chapter.

Open *theme-toggle.js* in a text or code editor. You'll notice it's completely empty. We'll be building this interface from scratch.

1. Use **document.createElement()** to create a **button** element:

   ```
   let $toggle = document.createElement( "button" );
   ```

2. Next, we'll label the button. Add the text "Theme Toggle" to the button by setting its **innerText** property (note that we also could have used the **createTextNode()** method with **$toggle**):

   ```
   let $toggle = document.createElement( "button" );
   $toggle.innerText = "Theme Toggle";
   ```

3. Finally, add the button to the **body**. You can use **querySelector()** to get this element, but there's also the convenient shortcut of **document.body** (which I've used here). Once you have the **body**, use **appendChild()** to add the button to it:

   ```
   let $toggle = document.createElement( "button" );
   $toggle.innerText = "Theme Toggle";
   document.body.appendChild( $toggle );
   ```

4. Save *theme-toggle.js* and open *toast.html* in your browser. You should see the button at the bottom of the page (FIGURE 24-9).

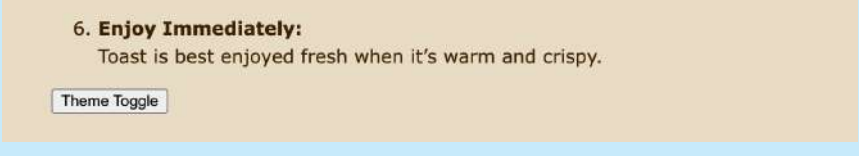


FIGURE 24-9. The **button** element you just inserted into the page

Congrats, you just conjured markup from the ether! You're a magician!

If you're writing a script that does anything with HTML, you're almost guaranteed to be working with attributes. For example, you might need to read the value of a form field to validate your user's response. The input's **value** is an attribute. Or maybe you want to disable your form's submit button until the user has checked the box stating they agree to your terms and conditions. You'll need to access the button's **disabled** attribute and the checkbox's **checked** attribute to do that. Regardless of the job you're looking to do with attributes, the DOM provides the tools to get it done.

There are several approaches to manipulating attributes and their values. In this section, we'll look at the **getAttribute()** and **setAttribute()** methods, accessing attributes via DOM properties, and the **classList** property that makes working with **class** attribute values more efficient.

The getAttribute() and setAttribute() methods

In JavaScript, we spend a lot of time "getting"—reading information from the page—and "setting"—adding or manipulating elements and their contents. Working with attributes is a big part of that. The first way the DOM enables us to work with attributes is via the well-named **getAttribute()** and **setAttribute()** methods. These methods are available on any HTML element node in the page.

To read the value of an element's attribute so you can do something with it later, call **getAttribute()** with a single argument: the attribute name. Include quotes, because the value must be a string:

```
element.getAttribute( "attribute_name" )
```

Let's look at an example using the markup for a simple image:

```
<img src="images/closed.png" alt="Closed Icon">
```

If I want to read its **src** attribute, I'll need to get a reference to the **img** element first, which I've done using **querySelector()** (see **Note**). Then I call the element's **getAttribute()** method, passing in the name of the attribute (**"src"**). When I log the result to the console, I can see that I get back the string value of the **src** attribute:

```
let $img = document.querySelector( "img" );
let closed_src = $img.getAttribute( "src" );
console.log( closed_src );  // "images/closed.src"
```

Now let's set a new value for the **src** attribute using **setAttribute()**. This method requires two arguments, the attribute to change and the new value for that attribute:

```
element.setAttribute( "attribute", "new_value" )
```

Using the same **img** markup, I can swap out the displayed image by changing the value of the element's **src** attribute. I start by defining a new **src** value for the image using the variable **open_src**. Then I can call **$img**'s **setAttribute()** method and pass in "src" (the attribute I'm setting) and that value (the new value for the **src** attribute) as the two arguments:

```
let open_src = "images/open.png";
$img.setAttribute( "src", open_src );
```

NOTE

Back in **Chapter 22**, *I explained that it's common for developers to use naming conventions to provide hints about what kind of value is associated with a variable. Here, I've used the* **$** *to prefix a variable containing an element reference (*`$img`*). The string value of its* **src** *attribute (*`closed_src`*) does not have that prefix, so I know it's not an element.*

■ YOU TRY IT

See if you can change the **alt** text for this image to be more appropriate for the new image, *open.png*.

Attribute properties

Many standard HTML attributes are also accessible as properties of an element. You access them using dot notation, referencing the property that corresponds to the attribute name you want to access. So, instead of using **getAttribute()** to access the image's **src** attribute, like this:

```
$img.getAttribute( "src" );
```

you can read the attribute's value using **$img**'s **src** *property*, like this:

```
$img.src;
```

The following would achieve the same goal we accomplished in the previous section using **getAttribute()** and **setAttribute()**, but more succinctly:

```
let $img = document.querySelector( "img" );
let closed_src = $img.src; // same as $img.getAttribute( "src" )

let open_src = "images/open.png";
$img.src = open_src; // same as $img.setAttribute( "src", open_src )
```

Knowing this, you're probably wondering why this wasn't the first approach I covered. Fair. It may seem simple, but this approach is a little trickier than you might think, because the property name doesn't always match the attribute name. For example, hyphenated attribute names need to be converted to camelCase to become properties (so, the **aria-hidden** attribute name is accessed through the **ariaHidden** property). That's because a hyphen is a reserved character used for subtraction in JavaScript. While not impossible to remember, this can trip you up.

The bigger issue, however, is when the property name is different altogether (see **Note**). The **class** attribute? It's **className**. Fun, right? The **for** attribute (used on a **label**)? It's **htmlFor**, because why not?! Again, these quirks aren't the end of the world, but they make using the properties a bit frustrating until you get familiar with them. I crave consistency, so I tend to gravitate toward using **getAttribute()** and **setAttribute()** most of the time. You may feel differently.

NOTE

The reason these two properties are named so strangely is that both **for** *and* **class** *are reserved words in JavaScript. The folks who designed the attribute property interface in the DOM opted for alternate names to avoid causing problems.*

Working with the class attribute

The DOM offers a *really* easy way to manage **class** names. That's handy, because you'll find yourself working with **class** names a lot in JavaScript; developers frequently use them to toggle interfaces between different states (e.g., closed versus open). The feature is called **classList**, and frankly, it's one of the most satisfying of the recent additions to the DOM. In this section, I'll show you why. You can follow along in the console if you have a browser handy, but I'll walk you through what's happening too.

The **classList** features are available for all elements, but we'll be using it with the **body** element in our examples. I'll start by getting a reference to it:

```
let $body = document.querySelector("body");
```

Now, let's explore some of the methods of this property.

classList.add()

First, let's add a **class** to the **body**. We use **classList.add()** to add a new **class** value ("test", in this case) to the **class** attribute:

```
$body.classList.add("test");
```

If the **body** doesn't already have a **class** attribute, this will add the attribute and assign the string value "test" to it:

```
<body class="test">
```

If the **body** element already has a **class** attribute, it will add the "test" value to the end of it. You'll recall that the **class** attribute takes a list of space-separated names, so this method appends the "test" value and ensures there's a space between it and any existing **class** names (see **Note**):

```
<body class="mobile homepage test">
```

NOTE

If the "test" **class** *already existed on the* **body** *element, this method is smart enough to avoid adding it a second time.*

classList.contains()

If I want to see if the "test" **class** is currently applied to the **body** element, I can use **classList.contains()**. It returns **true** or **false**, depending on whether the supplied **class** name exists in the string:

```
$body.classList.contains("test"); // true
```

classList.remove()

When I need to remove the "test" **class**, I use **classList.remove()**:

```
$body.classList.remove("test");
```

After calling this, the **body** will no longer have the "test" **class** applied. If "test" were the only **class** name applied to the **body**, the **class** attribute would also be removed:

```
<body ~~class="test"~~>
```

classList.toggle()

Finally, there's **classList.toggle()**. This is perhaps the unsung hero of the **classList** features. It adds the **class** name if it isn't already applied to the element, or removes it if it is already applied:

```
$body.classList.toggle("javascript-rocks");
```

Now that you have a better sense of how we can use the DOM to create elements and work with attributes, let's jump into EXERCISE 24-2 and add some attributes to the **button** element we created in EXERCISE 24-1. After that, we'll look at how to make it interactive with JavaScript events.

JAVASCRIPT EVENTS

In this section, I'm going to teach you how to make elements do things when your users interact with them. All interaction in JavaScript centers around something called **events**, so let's start by discussing what events are and how they work.

Thus far in our journey through the world of JavaScript, we've focused on writing code that runs once, when the page loads. That's not the only time we can run JavaScript code, however. You can write JavaScript that runs when users click a button, resize the browser window, or type something. This is how we build interactivity into our web pages.

EXERCISE 24-2. Adding attributes

In this exercise, you're going to take the **button** we created in EXERCISE 24-1 and set some attributes on it. Use the finished *theme-toggle.js* from the previous exercise as a starting point.

1. Open *theme-toggle.js* in a text or code editor. If you want to start fresh, a completed version of EXERCISE 24-1 is available with the materials for this chapter.
2. Immediately after the statement that created the button, set the button's **type** attribute to "button". You can use either **setAttribute()** or the element's **type** property, as I've done here:

```
let $toggle = document.createElement( "button" );
$toggle.type = "button";
$toggle.innerText = "Theme Toggle";
document.body.appendChild( $toggle );
```

3. I've already included some styles in the HTML document that will style the button to look like a switch control and move it to the top-right corner (or the top left, in right-to-left languages). If you'd like to take a peek at them, open up *toast.html*. To activate the styles, you'll need to add a **class** of "theme-toggle" to the button. Use **classList.add()** to do this:

```
let $toggle = document.createElement( "button" );
$toggle.type = "button";
$toggle.classList.add( "theme-toggle" );
$toggle.innerText = "Theme Toggle";
document.body.appendChild( $toggle );
```

FIGURE 24-10 shows the result. If you're wondering where the "Theme Toggle" label text went, I used CSS to hide it and to insert the "Light" label. If you view the page source, you'll see that it's still there. As a bit of a teaser, that text will switch to "Dark" when the dark theme is applied once you complete EXERCISE 24-4.

FIGURE 24-10. The button styled to look like a switch at the top of the page

Now that we have the button styled and in place, let's make it do something!

To understand how this works, you first need to understand that every interaction you have with a web page sends out a signal, called an **event**. Clicking on a link. Scrolling the page. Resizing the browser window. All events. You can think of them like little flares being fired off all over the place as people go about their business on the web.

We can write code that runs when these events fire. Most of the time when we work with events, we're writing code that runs when a specific event on a

particular element is fired off, such as when a user clicks (called a click event) on a search form's submit button.

Some of the most common events are **click**, **focus**, and **blur**:

click

A click event is fired off whenever you click with the mouse or touchpad, tap your finger on a touchscreen, or use the keyboard to activate buttons and links. Developers use it for all kinds of interactions, such as opening menus and navigating within carousels.

focus

The focus event fires when the cursor moves to an element, such as when you click into a form field or use the Tab key to move around the page. Developers often use it to help users see where they are in an interface.

blur

Blur is the opposite of focus; it fires when the cursor moves out of an element. Developers frequently use it on form fields to trigger code that validates a user's response, ensuring it meets the required format (e.g., is a valid email address).

For a great rundown on common events and what they do, check out the article "An Introduction to DOM Events" by Wilson Page (*oreil.ly/lQgOg*). For a complete list of JavaScript events, see the W3Schools reference guide (*w3schools.com/jsref/dom_obj_event.asp*).

Handling Events

We refer to code that runs when an event fires as an **event handler**. JavaScript offers two primary ways to handle events: property-based event handlers and event listeners.

Property-based event handlers

Property-based event handlers indicate the event to listen for as a property of the element. The property names start with "on" followed by the name of the event. For example, the click event is accessible as the `onclick` property, the focus event is accessed via `onfocus`, and so on.

In the following brief example, I create a button and add it to the page. Then I assign a function to the `onclick` property. That function will run when the event is fired:

```
// add a new button to the page
let $button = document.createElement( "button" );
$button.innerText = "Click Me";
document.body.appendChild( $button );
```

```
// respond to the click event
$button.onclick = function() {
    console.log( "Ouch!" );
}
```

You'll notice I haven't given the function a name. This is called an **anonymous function**. It's common to see anonymous functions as event handlers, as it's unnecessary to name the function if it's only being used for that one purpose.

You can also assign **named functions** as event handlers. Here's an example using focus and blur events (via the `onfocus` and `onblur` properties). When assigning a named function as an event handler, we use its name *without parentheses*, because we're not calling the function. The parentheses are only used after the function name when you're defining or calling the function. Otherwise, treat the function name like it's a variable, because you're referencing the function as a value:

Parentheses are only used after the function name when you are defining or calling the function.

```
function hi() {
  console.log( "Hi" );
}
function bye() {
  console.log( "Bye" );
}

$button.onfocus = hi;
$button.onblur = bye;
```

With these events assigned, when a user uses the Tab key to move around the page and they land on the button, the `hi()` function will run and log "Hi" to the console. When they use Tab again and move focus away from the button, the `bye()` function will run and log "Bye" to the console. Of course, in the real world, the function triggered by an event would do something more useful.

Developers typically use named functions as event handlers when they want to call the same function for multiple events. One such example is when they want to provide the same experience for mouse and keyboard users. To do that, they assign the named event handler to the click event and, when the button has focus, the activation key (typically Enter/Return or the space bar).

Event listeners

Property-based event handlers are great, but they have one important downside: you can only have one event handler per event. If you try to assign a second event handler, it overwrites the first one. To apply more than one event handler to an event, use the `addEventListener()` method.

This approach isn't all that different from creating a property-based event handler; it's just slightly more verbose to write. The `addEventListener()` method takes two arguments: the event name and the function you want to call when the event fires.

In this code sample, I've rewritten the property-based event handlers you just saw using **addEventListener()**:

```
$button.addEventListener( "click", function(){
  console.log( "Ouch!" );
});

$button.addEventListener( "focus", hi );
$button.addEventListener( "blur", bye );
```

As this code demonstrates, just like the property-based approach, **addEventListener()** allows you to use either anonymous or named functions as event handlers.

There's a lot more to JavaScript events than I've had time to impart to you here. When you're ready for more, check out Aleksandr Hovhannisyan's article "An Interactive Guide to JavaScript Events" (*oreil.ly/wSwWK*).

Now, let's put what you've learned into practice in EXERCISE 24-3.

NOTE

If you happened to run this code and the previous examples in the console and then played around with the button, you might have noticed that each event is now firing twice. That's because **addEventListener()** *doesn't replace property-based event handlers you've already assigned.*

Event Propagation

The relative simplicity of adding event handlers obscures what's going on under the hood when an event (such as a click event) fires. Behind the scenes, this event actually fires multiple times, in three distinct phases. We refer to these phases, collectively, as **event propagation**. FIGURE 24-11 shows a visual representation of these phases.

Here's what's happening in each phase:

- The first phase is the capturing phase. In this phase, the click event fires on the **window** object, then the **document** object, then the **html** element, then the **body**, and so on down the line—like a line of dominoes falling over—until it reaches the button you clicked, at which point it moves to the next phase.
- The second phase is the target phase. Here, the event is fired on the button you clicked. We call this the target element. Once the event is fired, the process moves on to the final phase.
- The final phase is called the bubbling phase. In this phase, the event works its way back out again, starting with the button's parent and ending when it reaches **window** again.

You can take advantage of event propagation to write more efficient and scalable JavaScript by listening for events further up the DOM tree (which is called **event delegation**). For more information on this topic, see "JavaScript Event Delegation: A Beginner's Guide" by Dmitri Pavlutin (*dmitripavlutin.com/javascript-event-delegation/*).

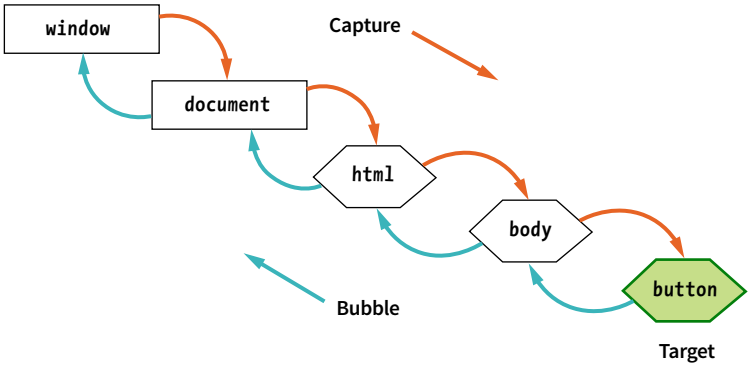


FIGURE 24-11. The three event phases in JavaScript

EXERCISE 24-3. Now it clicks!

In this exercise, you'll be adding an event listener to the button you created in EXERCISE 24-2 that toggles a **class** on the **html** element. I've already written the styles for how that class will alter the colors on the page, so you don't need to worry about that.

1. Open *theme-toggle.js* again. The finished version of the previous exercise is available with the materials for this chapter if you want a fresh start.
2. At the top of the file, create a variable (**$html**) to store a reference to it. You can use **querySelector("html")** or use the shortcut to it (**document.documentElement**). We're declaring this variable at the top of the document so it's available to all the code that follows (see **Note**).

```
let $html = document.documentElement;
let $toggle = document.createElement("button");
// ↓ existing code
```

NOTE

document.documentElement *points to the* **html** *element in the same way* **document.body** *points to the* **body** *element.*

3. Now, move to the end of the file and make some space just before you call **appendChild()** to insert the button. We're going to put our event-related logic here. Start by declaring a new function, which we'll use to toggle between the light and dark themes. You can name it whatever you like. (If you need help coming up with a good name, refer to the section on naming functions in **Chapter 23, Functions, Conditionals, and Loops**.) I chose to call mine **toggleTheme()**:

```
// ↑ existing code
$toggle.innerText = "Theme Toggle";
function toggleTheme() {

}
document.body.appendChild( $toggle );
```

4. Inside that new function, use **$html**'s **classList.toggle()** method and pass in the string "dark". Remember, this method will add "dark" to the **class** if it's not already there or remove it if it's already assigned:

```
function toggleTheme() {
  $html.classList.toggle( "dark" );
}
```

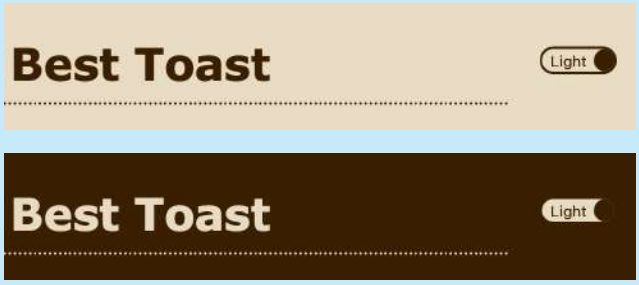


FIGURE 24-12. The button is now clickable and will toggle the site between a light theme and a dark theme.

5. Finally, we'll listen for when the user clicks on the button, then use the **toggleTheme()** function (or whatever you named it) as the event handler. After the function definition, assign the function as the event handler on the button for the click event. You may use the **onclick** property or **addEventListener()**, depending on your preference. I used the **onclick** property since we only need one event assigned to the button. Remember, you don't include the parentheses when using a named function as a value; you only include them when you are either defining a function or calling it.

```
function toggleTheme() {
  $html.classList.toggle( "dark" );
}
$toggle.onclick = toggleTheme;
```

6. Save the file *theme-toggle.js* and open *toast.html* in your browser. Now when you click the button, the page will shift from the light theme to the dark theme and vice versa (see FIGURE 24-12). The button still says "Light" regardless of the theme applied, but we'll fix that in a moment. Great work!

REFACTORING FOR ACCESSIBILITY

As you saw in FIGURE 24-12, the button's label doesn't change to "dark" when the dark theme is applied. It would be good if it did that. There's also no indication of exactly what the control does. Some users may be able to figure it out, but most folks probably won't know what to expect. And it's not just the average user who is likely to be confused by this...people who rely on assistive technologies like screen readers are going to have no idea what to make of it.

We've got some work to do to improve this UI, so let's tuck in.

Accessibility in JavaScript

You were first introduced to the concept of accessibility in **Chapter 3**, and we've been keeping it in mind throughout this book. There are measures we need to take to ensure the accessibility of the features we build that rely on JavaScript as well.

It's incumbent on us to use markup that enables people who rely on assistive technologies to understand and interact with what we've created. As it stands right now, we've utterly failed them. They will see the label—"Toggle Theme" that we set as the button's text content in EXERCISE 24-1—but it's mediocre at best. There's also no indication of what state it's in or what will happen when you click it. Thankfully, we have ARIA (Accessible Rich Internet Applications) attributes that help make our interactive features more accessible (refer back to **Chapter 5, Marking Up Text**, for a refresher). Among the many attributes ARIA offers, two are perfect for our situation: **`aria-pressed`** and **`aria-label`**.

aria-pressed

The **`aria-pressed`** attribute is part of a subgroup of ARIA attributes that indicate the *state* of an interface. They help users know, for example, whether or not a checkbox is checked (**`aria-checked`**) or if a field is read only (**`aria-readonly`**). The **`aria-pressed`** attribute was designed for toggle buttons like ours. It takes a Boolean string value of "true" or "false".

Here's an example of this attribute applied to an HTML pause button:

```
<button aria-pressed="false">Pause</button>
```

This indicates, in an accessible way, whether the pause button is currently pressed. In this case, it's not, because **`aria-pressed`** is set to "false". It's important to note that this attribute does not get updated automatically when the user clicks the button. We need to use JavaScript to change its value from "false" to "true" and back again, keeping it in sync with the current state of the button.

The **aria-pressed** attribute isn't just useful for accessibility, though. Developers also reference it to determine the current state of the button when other parts of the program are running. To continue with the media playback theme, you could imagine a shuffle button that controls whether the songs being played should be shuffled. When a song ends and the program is getting ready to select the next song, it can reference the shuffle button's **aria-pressed** state to determine if it should go to the next song or randomly pick a different one. Developers also use this attribute as a styling hook—using an attribute selector such as **[aria-pressed="true"]**—to alter the appearance of the button depending on whether it's pressed or not.

aria-label

The **aria-label** attribute enables us to label an element in a way that is invisible to sighted users but available to assistive technologies like screen readers. Think of it like the **alt** attribute, but for interface items. Here's an example that uses two vertical pipes for a pause symbol, but is properly labeled using **aria-label**:

```
<button aria-pressed="false" aria-label="Pause">||</button>
```

When you make changes to an interface, it's important to update the **aria-label** attribute to properly reflect the current state of things. The previous example shows the resting state of the button: it's not pressed and is labeled "Pause." When a user clicks it and the program updates the **aria-pressed** value to "true", the **aria-label** value also needs to get updated to reflect the current state, "Paused":

```
<button aria-pressed="true" aria-label="Paused">||</button>
```

Setting ARIA attribute values

As you saw earlier in this chapter, setting values for attributes is straightforward. You can use **setAttribute()** if that's your jam, or you can use the corresponding HTML properties **ariaPressed** and **ariaLabel**, respectively.

The following examples illustrate the two different approaches. To keep it brief, I've skipped creating the **$button** variable. Assume it references the pause button from the previous section.

```
// DOM method
$button.setAttribute( "aria-pressed", "true" );
$button.setAttribute( "aria-label", "Pause" );

// HTML property
$button.ariaPressed = "true";
$button.ariaLabel = "Pause";
```

Now that you have a handle on how we can use ARIA attributes to improve the overall accessibility (and usability) of the UIs we create, let's refactor our code to make it useful to more people in **EXERCISE 24-4**.

The title Attribute

For a long time, the **title** attribute was thought to be great for accessibility. It's not. Pretty much all assistive technologies ignore it because, back in the day, developers used to stuff it with keywords hoping to boost their search engine rankings. It got to the point where exposing the contents of **title** to screen reader users was doing more harm than good. That doesn't mean it has no value, however. The **title** attribute can provide a built-in tooltip when users linger over an element with their mouse, which can be useful to some folks (see **FIGURE 24-13**).

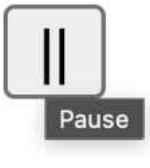


FIGURE 24-13. The Pause button with a tooltip provided via the **title** attribute

EXERCISE 24-4. Accessibility refactor

In this exercise, you'll create a new function to maintain the proper state and label on the toggle button. The values you set will be determined by whether the **html** element has the "dark" **class** applied to it.

1. Open *theme-toggle.js* again. If you want to start fresh, there's a completed version of EXERCISE 24-3 in the chapter's resources.
2. We're going to start by creating a function that will manage the state of the button by setting the appropriate labels (**aria-label** and **title**) and **aria-pressed** state. At the top of the file, just below the line declaring the `$html` variable, add a little space and create a new function. Call it **updateButton()**:

```
let $html = document.documentElement;
function updateButton() {

}
// ↓ existing code
```

3. The first thing we will need to figure out is whether the **html** element (`$html`) has the "dark" class applied. Create a new variable to track whether it's in dark mode or not. Use **classList.contains()** to assign its value, which will be **true** or **false**:

```
function updateButton() {
  // is the dark theme currently applied?
  let dark = $html.classList.contains("dark");
}
```

4. Let's take care of setting the right value for **aria-pressed** on the toggle button (`$toggle`) next. Remember, this property tells assistive technology (AT) users the current state of the toggle button. The *toast.html* document includes predefined styles that will apply based on this value.

   We can use a conditional (**if...else**) to help us assign the appropriate value to the attribute. Since the value returned by **classList.contains()** is a Boolean (**true** or **false**), we can use the stored variable **dark** on its own in the test condition. In other words, in this case, there is no need to write out **if (dark === true)** because we get a **true** or **false** value from **classList.contains()**.

   We'll assign the value "true" if dark mode is applied (i.e., the user has clicked the button and toggled on dark mode) and "false" if it's not (if light mode is currently applied). We'll use the **ariaPressed** property to do this, but you can use **setAttribute()** instead if that's your preference:

```
function updateButton() {
  let dark = $html.classList.contains("dark");
  // update the aria-pressed state of the button
  // for AT users and for the right CSS to apply
  if ( dark ) {
    $toggle.ariaPressed = "true";
  } else {
    $toggle.ariaPressed = "false";
  }
}
```

5. Next, to help people who rely on screen readers know what this button will do, we're going create a better label and update its value as appropriate. For it to be a useful label, we should provide details about what will happen when a user clicks the button. I chose "Change theme to X," where X will be the name of the other theme. Create a variable, **label**, inside the function and set that string as its value:

NOTE

I have added comments in the examples to help you understand what's happening at each step. You do not need to copy the comments into your own code.

EXERCISE 24-4. Continued

```
function updateButton() {
  let dark = $html.classList.contains("dark");
  if ( dark ) {
    $toggle.ariaPressed = "true";
  } else {
    $toggle.ariaPressed = "false";
  }
  // set the default label
  let label = "Change theme to X";
}
```

6. Now, at the top of the function, before the **`if...else`** conditional, create another variable (**`other_theme`**) to track the name of the theme that will apply when the button is clicked. If the dark theme is applied, its value should be "light"; when the light theme is applied, it should be "dark". Declare the variable first and then use the condition we created in step 4 to set the appropriate value:

```
function updateButton() {
  let dark = $html.classList.contains("dark");
  // track the name of the other theme
  // so we can use it in the label
  let other_theme;
  if ( dark ) {
    $toggle.ariaPressed = "true";
    other_theme = "light";
  } else {
    $toggle.ariaPressed = "false";
    other_theme = "dark";
  }
  let label = "Change theme to X";
}
```

7. Next, replace the placeholder "X" from step 5 with the variable you just created. The easiest way is to delete "X" and concatenate `other_theme` with the string value:

```
// make the label differ based on the theme
let label = "Change theme to " + other_theme;
```

8. The last step in this function is to assign the value of **`label`** to the toggle button's **`aria-label`** and **`title`** attributes. I've used the attribute properties—**`ariaLabel`** and **`title`**—to do this, but you could also use **`setAttribute()`**:

```
function updateButton(){
  let dark = $html.classList.contains("dark");
  let other_theme;
  if ( dark ) {
    $toggle.ariaPressed = "true";
    other_theme = "light";
  } else {
    $toggle.ariaPressed = "false";
    other_theme = "dark";
  }
  let label = "Change theme to " + other_theme;
  // apply the label to the aria-label
  // and title attributes of the button
  $toggle.ariaLabel = label;
  $toggle.title = label;
}
```

EXERCISE 24-4. Continued

9. At this point, you've made a robust function for managing the values of **aria-pressed**, **aria-label**, and **title** for this UI. Now, it's time to use it. To do this right, we need to call the `updateButton()` function in two places. The first place we'll call it is at the end of the `toggleTheme()` function we created in EXERCISE 24-3. This will keep the button updated as things change:

   ```
   function toggleTheme() {
     $html.classList.toggle( "dark" );
     // update the button state when it's clicked
     updateButton();
   }
   ```

   We also need to update the button before we insert it onto the page. To do this, call the `updateButton()` function just before you inject the button into the page using **appendChild()**.

   This will ensure the right values are applied immediately when the script loads, before the user can access it:

   ```
   // ↑ existing code
   $toggle.onclick = toggleTheme;
   updateButton(); // sets the initial state
   document.body.appendChild( $toggle );
   ```

10. Save the *theme-toggle.js* file and refresh *toast.html* in your browser. You should now see a tooltip when you mouse over the toggle button, and you should see the **aria-label** attribute if you inspect the element using your browser's developer tools. If you click the button, you should also see it change, because we wrote the code to change the **aria-label** and **title** when the button is pressed. FIGURE 24-14 shows what it looks like in the two different states.

This toggle has really come together. Give yourself a pat on the back!

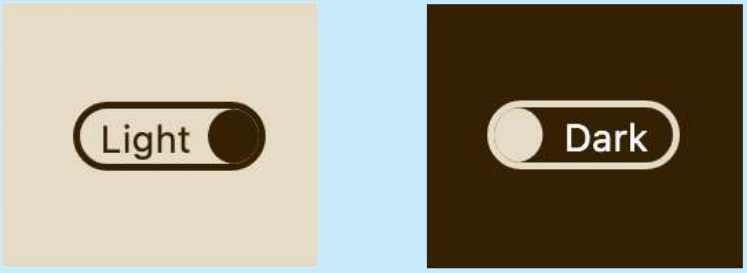


FIGURE 24-14. Two different states for the toggle button. The different styles are applied based on the [**aria-pressed="true"**] selector.

Web Storage

In EXERCISE 24-4, we improved the usability and accessibility of the theme toggle interface. One drawback of what we've built so far is that refreshing the page resets the theme. That's not ideal. If a user chooses a particular theme, we should honor that and display their chosen theme when they return to the page (or, to use developer lingo, their choice should **persist**).

The easiest way to accomplish this goal would be to store the current theme in the user's browser so we can retrieve it when the page is loaded. One option for this is to use the browser's `localStorage` object (a feature of all modern browsers).

The `localStorage` object allows you to store information in **key/value** pairs. You can think of a key as a label for the information (similar to a variable name) to which you assign a string value. For example, you could assign the value "Hi!" to a key you named "greeting". What makes `localStorage` different from variables is that information you put in `localStorage` will be available on every page of your site and persist between page refreshes.

If you'd like to take the theme toggle to the next level, I've prepared an article on `localStorage` with a bonus exercise to guide you, which is available at *learningwebdesign.com/articles*.

WRAPPING UP

Pause for a moment and think about everything you accomplished in this chapter. You conjured markup out of thin air. You controlled which styles are applied to a page by manipulating `class` attributes. You also put in the work to improve the accessibility of your code, which, frankly, makes you one of the best developers on the web today.

Take a bow, that was amazing! You should be proud of your progress. If you're up for another challenge, check out EXERCISE 24-5.

EXERCISE 24-5. Challenge yourself!

You've picked up some amazing skills in this chapter. Let's see if you can push them a little further. The code I wrote to tackle this challenge is provided with the materials for the chapter, if you want to see how I solved it.

You can do this exercise in *theme-toggle.js* if you like, or directly in the console. Create a list—`ol` or `ul`, it's your choice—with at least two items in it, using only JavaScript. The tools you'll employ are `document.createElement()`, `innerText`, and `appendChild()`. Combine the elements together *before* adding the list to the page to reduce the number of times the browser needs to repaint the page.

TEST YOURSELF

I've put together the following questions to test how well you've retained what you learned in this chapter. Answers are available in **Appendix A**.

1. What is the development philosophy that reminds us to build the most basic interface first and then improve it as we can?

   a. Graceful degradation

   b. Inclusive design

   c. Universal design

   d. Progressive enhancement

2. What method do you use to make new markup with JavaScript?

   a. `document.createElement()`

   b. `document.newElement()`

   c. `document.HTMLElement()`

   d. `document.createHTML()`

3. Which of the following are built-in methods that insert markup into the DOM?

   a. `insertAfter()`

   b. `insertBefore()`

   c. `prependChild()`

   d. `appendChild()`

4. If you wanted to add a **`class`** of "dark" to the **`html`** element assigned to the variable `$html`, which of the following statements would work?

   a. `$html.className = "dark";`

   b. `$html.setAttribute("class", "dark");`

   c. `$html.classList.add("dark");`

   d. `$html.classList.toggle("dark");`

5. Of the following statements, which would set an event handler on `$input` that reacts to changes in the input's value?

   a. `$input.onchange = someFunction;`

   b. `$input.onchange = someFunction();`

   c. `$input.addEventListener( "onchange", someFunction );`

   d. `$input.addEventListener( "change", someFunction );`

6. Match the event phase to its definition for a click event on `$button`:

   ___ Triggers the event on every ancestor of `$button` from **`window`** to **`document`** on down to its parent

   ___ Triggers the event on `$button`

   ___ Triggers the event on every ancestor node from `$button`'s parent up to and including **`window`**

   a. Bubbling
   b. Targeting
   c. Capturing

7. What do we call the special set of accessibility-focused attributes?

   a. AIRA
   b. RIAA
   c. ARIA
   d. AARI

8. What attribute do we use to indicate the state of a toggle button?

   a. `aria-toggled`
   b. `aria-pressed`
   c. `pressed`
   d. `checked`

9. Which attribute can you use to provide an invisible label for assistive technology users?

   a. `title`
   b. `aria-label`
   c. `aria-description`
   d. `hidden-label`

10. How should you use the **`title`** attribute?

    a. To provide an accessible label
    b. To provide SEO keywords
    c. To provide the title of the page
    d. To provide a tooltip to sighted users

CHAPTER
25

NEXT-LEVEL JAVASCRIPT

by Aaron Gustafson

IN THIS CHAPTER

JavaScript libraries

Web components

Frameworks

Using JavaScript outside the browser

Progressive web apps

By this point, you have a solid foundation in what JavaScript is, how it works, and how to use it. You know how to create variables and functions. You understand many of the different value types in JavaScript, such as strings, numbers, Booleans, arrays, and nodes. You know how to generate and alter HTML markup. You also know how to enable interactivity using JavaScript events.

Even with all the knowledge you've gained thus far, we've only just scratched the surface when it comes to JavaScript. I want to take this opportunity to introduce a handful of more complex topics you should know about if you plan to pursue a career in web development. The topics I'll be covering in this chapter are considerably more advanced than where we left things at the end of **Chapter 24, Working with the DOM and Events**, but they will help you get a sense of the directions you could take your career in if you've been bitten by the JavaScript bug.

We'll begin this grab bag of a chapter by discussing JavaScript libraries and how to integrate them. We'll follow that with an introduction to web components as a way of making JavaScript user interfaces more portable between projects. Then we'll shift gears to talk about JavaScript frameworks as ready-built sets of tools for rapidly building large web projects. Next, we'll explore how you can use JavaScript outside the browser context. Finally, we'll round out the chapter with a discussion of progressive web apps and how to enable your web projects to function more like traditional apps.

Let's dig in!

JAVASCRIPT LIBRARIES

Once upon a time it was impossible to find elements in the DOM using CSS selectors, so an enterprising developer wrote a function to fill that gap and make it easier to find the elements they needed to work with in the page. They started out using this function on a single project. Then another. Then another. Over time, they refactored it to make it run faster. Eventually, it and a handful of other useful functions ended up in a JavaScript file they reached for at the start of every new project. Without realizing it, they became the proud owner of a JavaScript library.

A **library** in this case isn't a collection of books and journals; it's a collection of JavaScript code—typically functions—that can be reused for common tasks. There are countless libraries of JavaScript code out in the world. They are created and maintained by developers who've identified gaps and filled them with their own code. Some are used only by a single developer (or team) for a single project. Others grow to the point where developers around the world are using and contributing to them at such a scale that they become ubiquitous on the web (at least for a time).

History has proven that most JavaScript libraries have a shelf life. That's not because they aren't good—many are amazing—but because the best ideas from popular libraries are frequently integrated into the JavaScript language itself and how it accesses the DOM (see the sidebar **"The Road to Standardization"**). And so, over time, these custom libraries become less useful, and the standard JavaScript library gets better and better.

As you become more familiar with the JavaScript language and its capabilities, you'll likely find you need to do something that the language itself or the DOM just doesn't support. Or you may encounter a feature that's unintuitive or clunky to use and come up with a better way to work with it. You might end up with a JavaScript library of your own. And who knows—your code could even influence the standard!

The Road to Standardization

This is exactly what happened with Simon Willison's custom `getElementsBySelector()` function, written in 2003. It inspired a core function of another library called jQuery, which was released in 2006. JQuery became incredibly popular—at one time, it was being used on over 70% of websites, including those belonging to some of the biggest companies in the world. Its popularity drove interest in standardizing the feature, which ultimately led to the inclusion of `document.querySelector()` and `document.querySelectorAll()` in Safari in 2009, with other browsers quick to follow. And this is just one of many examples how developers have influenced the continued evolution of JavaScript.

Finding and Using JavaScript Libraries

While it's entirely possible for you to roll your own library over time, it's also likely you'll find yourself wanting to leverage the work of other amazing developers who've devoted countless hours to making it easier for you to do things with JavaScript.

For example, if you find yourself needing to work with dates and times in JavaScript, you will undoubtedly go gray at least six months sooner than you might otherwise. If, frustrated with JavaScript's cumbersome approach, you ask your trusty search engine "How can I make it easier to work with dates in JavaScript?" you'll get some useful suggestions. Chances are, someone else has come up against the same (or a similar) limitation. Sometimes you'll find

code you can adapt to your needs; other times you might find that the code you're looking for is part of a larger library of JavaScript code.

Once you find what you're looking for, you can incorporate JavaScript library code in one of three ways:

Copy the code

If you're permitted (we'll discuss licensing in an upcoming section), you can copy the code from someone else's library and incorporate it directly into your site's JavaScript code. Developers typically take this approach when they only need a portion of the library.

Link to the code

Many JavaScript libraries let you download an optimized JavaScript file directly from the library's website or GitHub repository. Once you grab that file, you can drop it into the *js* directory in your website and link to it using a `script` element. Some libraries also allow you to link directly to files hosted publicly on their own content distribution networks (CDNs) or aggregators like UNPKG (*unpkg.com*) or jsDelivr (*jsdelivr.com*).

Import the code as a module

A lot of modern JavaScript is distributed in a format called a module (see the **"JavaScript Modules"** sidebar). The library code still exists in a file (or sometimes several files), but instead of using a `script` element to link to it, you use an `import` statement to incorporate the specific functions you want to use in your code.

The Choice May Not Be Yours

If you join a team, it's entirely possible they will have already adopted a set of libraries and maybe even a framework. To do your job effectively, you'll need to learn how to use them. In that case, the steps I outline in this section will be a helpful starting point for learning about the tools being used by your new team.

Popular JavaScript Libraries

The following are some of the JavaScript libraries you may encounter during your development career (see **Note**):

jQuery (*jquery.com*)

At the time I'm writing this, once-dominant libraries like jQuery are waning in popularity, but this is still an active project. It's a small library (about 30 KB, compressed; the size of a small image file), and many designers and developers love how simple it makes building out many kinds of interactions. There are also thousands of community-developed jQuery plug-ins (small, single-purpose scripts that leverage jQuery) that make it easy to add rich functionality in little to no time.

GSAP (*gsap.com*)

GSAP is an animation library. If you want to make your web pages animate like a movie title sequence or a fancy slideshow, this is the library you'll be reaching for.

NOTE

React.js calls itself a library, but because it does more than the typical library, I am covering it in the upcoming **"JavaScript Frameworks"** *section.*

JavaScript Modules

JavaScript modules enable you to manage your code in separate files and pull it together as needed to build your program. Similarly, if you need just one function from a larger JavaScript library, JavaScript modules give you the freedom to load only the one piece you need. This approach is relied on heavily in JavaScript frameworks and backend JavaScript (topics we'll discuss later in this chapter), so it's a useful concept to understand.

There are two different syntaxes for writing JavaScript modules: CommonJS and ECMAScript Modules (ESM). CommonJS, the older syntax, is only really used in legacy projects today, so it's not very, well, common. You're much more likely to encounter the newer ESM syntax, so I'll focus on that.

To illustrate how modules work, I'm going to create a file called *hey-there.js*. Its contents will be a single function called `sendGreetingsFrom()`:

```
// hey-there.js
export function sendGreetingsFrom( name ){
  console.log(`Hello from ${name}`);
}
```

For the most part, this should look straightforward. You might be wondering what the **`export`** piece is all about, though. The **`export`** keyword is how we make code from this file (which is a module) available to other files. Everything you export—functions, variables, etc.—becomes public code; anything you declare in the file but don't export remains private to the module. Your public functions within the module can reference or even provide access to the private stuff, but it's not otherwise accessible from outside the module.

In this case I'm making the **`sendGreetingsFrom()`** function available, but I could just as easily have exported any other JavaScript value type (strings, numbers, objects, etc.), or even multiple values. Because I'm exporting it as a named function, we call it a **named export**.

To use this exported function in another file (*main.js*), I *import* it with an **`import`** statement, like this:

```
import something from module
```

The *`something`* here refers to what you want to get from the module. Often it's one or more named exports, surrounded by a set of curly braces (as you'll see in a moment). The *`module`* piece is the location of the JavaScript module you want to import from. It can also be a reference to a JavaScript package, which we'll discuss later in this chapter (in **"JavaScript Beyond the Browser"**). Because the two files in this example exist in the same directory, my import is simply:

```
// main.js
import { sendGreetingsFrom } from "./hey-there.js";
```

This imports only the named export **`sendGreetingsFrom`** from the *hey-there.js* file. Once it's imported, I can use the function as if it had been declared within *main.js*:

```
// main.js
import { sendGreetingsFrom } from "./hey-there.js";
sendGreetingsFrom("Aaron"); // "Hello from Aaron"
```

This approach to writing JavaScript is quite powerful and makes it incredibly simple to author code that's easy to port between projects. It also makes it easy to work with code that other folks have written.

Luxon (*moment.github.io/luxon*)

Remember when I told you working with dates was challenging in JavaScript? Luxon makes it so much easier.

Three.js (*threejs.org*)

If you need to create and render 3D graphics on the web, there's a good chance you'll be using Three.js to do it.

D3.js (*d3js.org*)

This the go-to library for data visualizations. It has a steep learning curve, but you can do some pretty amazing things with it.

For additional library resources, see "The 40 Best JavaScript Libraries and Frameworks" by Durga Prasad Acharya (*kinsta.com/blog/javascript-libraries/*). MDN provides a good jumping-off point for various library tutorials at *oreil.ly/wd1ls*.

Things to Consider When Selecting a JavaScript Library

Depending on your role, you may not always have the option of choosing which library to use. Still, it's important to understand how to compare options to select the best fit for your project.

Every library has a different philosophy and community behind it. While it may seem cut and dried—*I need to do X and this library does that*—the reality is that there are factors beyond the code that you should consider. The following questions can help you make the best decision:

Is this library actively maintained?

There are a ton of JavaScript libraries, tools, etc. that are showing their age or that were created with great enthusiasm only to be abandoned when the developer(s) moved on. If the code is no longer maintained, it may not work anymore or may not be the most efficient way to accomplish your goal. You'll need to be prepared to maintain the code yourself if you adopt it.

How's the support for this library?

Is there an active discussion board or community server for the project? Are folks on there helping each other out with problems? Who do you contact if you have issues? If you run into a problem, you'll want to know where you can go for help. If there's no support available, that's important to know too, as you'll need to find your own solutions.

How is the code licensed?

Many JavaScript libraries are licensed under the liberal MIT License, which allows you to use them freely for both personal and commercial projects. Others aren't. Some licenses require you to release your entire codebase as open source if you incorporate the code they provide. That

■ CONSIDER THIS

The Standard Library

The JavaScript standard itself, with all its built-in methods and properties, can be thought of as a type of library. It's full of existing tools you can use in your scripts.

may be cool with you (or your managers), or it may not. Read the fine print, and consult with a lawyer if you don't understand what the license requires of you.

Is the code in this library robust and accessible?

Lots of folks can write JavaScript and chuck it on the web for others to use. That doesn't mean that it's quality code. If the library is on GitHub (or similar), check the list of outstanding issues for open accessibility bugs or performance-related problems.

Is the code doing only what you need?

Read the source. Make sure you understand what the code is doing for you (to the extent you can). Look for signs that it could be doing nefarious things like sending personal data from your website to another server.

Is the library doing too much?

Many libraries are chock-full of utilities. Chances are you won't need them all. Most libraries enable you to extract only the bits you need, which will help you reduce the amount of JavaScript code in your project, but not all libraries are set up or licensed for that kind of use. Look for open source projects that offer their code under the MIT License, as it grants full permission for you to use their code and adapt it to your needs.

Slowing down and running through this checklist can help you make good choices when adopting JavaScript libraries you find on the web. These are also good points to consider from the flip side if you ever consider releasing some of your own code for others to use in their projects.

There used to be two popular kinds of JavaScript libraries: libraries of utilities that make it easier to do specific kinds of work (like most of the ones I mentioned in this section) and libraries of UI components that you can drop into your projects easily. The utility-focused libraries are here to stay. They do one job and do it well, and the best ones play nicely with other libraries as well as frameworks. The more UI-focused libraries are starting to lose ground to an emerging standard that we'll explore in the next section: web components.

WEB COMPONENTS

As you build more and more websites, you'll frequently find yourself needing to create user interfaces that do largely the same thing again and again. The theme toggle we built in **Chapter 24** is a good example of this. Just as JavaScript libraries allow us to reuse the same JavaScript code on different sites, **web components** do the same thing, but using a combination of JavaScript and HTML. Web components are technically a collection of technologies that allow you to create custom, reusable, single-purpose elements

for building web pages and interfaces (see the **"Web Component–Related Standards"** sidebar).

Imagine I'm working on a sign-up form for my web store. The form includes a password field and a password confirmation field. The values a user types in those two fields need to match so I know they've entered their password correctly. I could write some JavaScript to ensure that requirement is met, include that in my JavaScript file, and call it a day—but that approach won't be easy to reuse on another site, especially if the script is dependent on specific field names and such.

An alternative would be to create a **custom element** named **`require-matching-values`** that requires any two fields within it to have the same value. A custom element is an HTML element you get to define. The JavaScript logic would be embedded in the custom element, allowing me to avoid references to specific field names. All I'd need to do to implement the feature is use the element in my HTML file:

```
<require-matching-values>
  <label for="password">Password
    <input type="password" name="password" id="password">
  </label>
  <label for="password-confirm">Confirm Password
    <input type="password" name="password-confirm" id="password-
confirm">
  </label>
</require-matching-values>
```

Web components come in a variety of flavors and can get quite complex, depending on what you want to do with them. I've even seen a whole site built as a single web component, though I don't recommend it! While this is a topic that is too massive to really tuck into in this chapter, I do want to give you a little taste of one way you can use them.

Let's say I wanted to make the theme toggle from **Chapter 24** into a web component. To do that, I need to create a custom element. You can name a custom element whatever you like, so long as the name includes a hyphen (-). That requirement exists to keep custom element names from conflicting with the names of standard HTML elements, now or in the future. For consistency, I've called the custom element **`theme-toggle`**.

The first step in creating this custom element is to adapt the JavaScript code we wrote in **Chapter 24** into a JavaScript class. A **JavaScript class** is a recipe for building a JavaScript object. Just as you might write up instructions for how to make your favorite snack to share with a friend, you can use JavaScript to provide instructions for how to create a particular type of object (see **Note**).

Without going into too much detail, I'll share a quick overview of the process. I start by defining a new element class called **`ThemeToggleElement`** that *extends* (builds upon) the base **`HTMLElement`** class that is already part of the DOM. By convention, we name custom elements using title case (capitalizing

Web Component–Related Standards

Web components represent the standardization of techniques made popular in libraries and frameworks. As such, there is no single web components specification; rather, web components rely on four other specifications:

- The Custom Elements specification
- The Shadow DOM specification
- The HTML Template specification
- The ES Module specification

To explore these specifications and how they work together, ***WebComponents.org*** is a good place to start.

NOTE

*If you need a refresher, an object is a JavaScript value that has its own properties (i.e., variables) and methods (i.e., functions). Many objects are defined in the JavaScript standard and the DOM, but you can also define your own objects. See **oreil.ly/-16EC** for more information.*

each word) and end their names with "Element." So, the **theme-toggle** custom element maps to the **ThemeToggleElement** class in JavaScript:

```
class ThemeToggleElement extends HTMLElement {
  // All the code for the theme toggle goes here
}
```

Within that class, I put the code necessary to make the theme toggle work. The code we wrote requires a slight refactor to work in the context of a JavaScript class, which is a bit different from the functions we used in the exercises, but the changes won't be substantial.

Below that class definition, I call **customElements.define()** to connect this class to the custom element name **theme-toggle**. Before calling that method, however, I need to ensure the **customElements** feature exists, which I can do using a conditional **if** statement:

```
if ( window.customElements ) {
    customElements.define(
      "theme-toggle", ThemeToggleElement
  );
}
```

What makes the web component approach powerful is that it fully encapsulates the code for the theme toggle into a single, portable JavaScript file. To use it another project, I can just drop in the **theme-toggle** element and include the JavaScript file containing the custom element definition somewhere in the document:

```
<theme-toggle></theme-toggle>

<!-- elsewhere in your HTML -->
<script src="theme-toggle.js"></script>
```

If you're curious to see what a web component version of the theme toggle looks like, I've included that with the resources for this chapter.

Further Reading

As I mentioned, this has just been a quick introduction to web components and custom elements. It's a huge topic, but here are some good places to start if you're eager to learn more:

- "About Web Components" by EisenburgEffect (*oreil.ly/w_IOe*)
- "Custom Elements v1—Reusable Web Components" by Eric Bidelman (*web.dev/articles/custom-elements-v1*)
- "How to Build Your First Web Component" by Joe Attardi (*oreil.ly/FLMSA*)

JAVASCRIPT FRAMEWORKS

Many developers (and teams) have gone all-in on JavaScript, using it as the backbone of all their development efforts. They use JavaScript to generate their HTML and CSS, handle navigation from page to page (which they call **routing**), connect to a database, handle logins, translate their site into different languages, and so much more.

The scale of what they are doing with JavaScript goes far beyond the scope of a traditional JavaScript library. These development environments that can handle a much wider scope of site development are known as **frameworks**.

To use a metaphor, imagine you needed to build a house. You could build it from the ground up, mixing and pouring the cement for the foundation, cutting up the lumber to erect the walls, and so on. Or you could buy a prefab kit for a house and assemble it according to a set of instructions. You'd be a little more limited in how customized the final house would be because you're working with prebuilt parts, but you'd probably get it up a lot faster.

You can think of JavaScript frameworks as providing the prefab parts you can use to quickly build out a complex website. Adopting a framework requires you to make some sacrifices when it comes to customization and performance (which I'll discuss below), but frameworks can absolutely help get you further, faster.

Building with Components

Nearly all modern JavaScript frameworks start from the premise of a web page being composed of a collection of **components**. Those famous plastic bricks you probably played with as a kid provide a good analogy here. Just as you might use differently shaped plastic bricks to build a house, a car, or a house/car hybrid, you stick components together to build out your web pages.

At its most basic, a component within a JavaScript framework is a discrete piece of reusable code that produces HTML. Depending on the component, that code might also include CSS to handle the visual design and additional JavaScript to govern the component's behavior. A component could be as simple as a button or a link or as complicated as your site's navigation or a form.

In addition to using components to build pages, you'll often create more complex components from simpler ones. What do I mean by that? Here's a hypothetical component hierarchy that would produce a navigation menu, starting with the larger component and then drilling down:

The* `MainNavigation` *component

This component gets loaded into every page in your site. It defines the list of pages that will be shown in the navigation. In the component's logic,

it loops through those links and inserts a `NavigationItem` component for each one, passing it the display text and its URL.

The `NavigationItem` component

This component expects to be passed a text label and a URL. The logic in the component checks to see if the URL that was passed in matches the current page. It then inserts a `Link` component and passes the text label, the URL, and an indication of whether this is the current page to it.

The `Link` component

This component expects a text label, a URL, and an indication of whether it's the current page. It takes all that information and builds an anchor element (`a`) with the URL as its `href` and the text label as its text content. If it's told this link points to the current page, it also inserts the attribute `aria-current="page"` so assistive technology users are informed and the appropriate styles can be applied.

Each component has a single responsibility, making code easier to write, maintain, and test.

It may seem like overkill to break things down this way, but it allows each component to have a single responsibility. That makes the code easier to write, maintain, and test to ensure it's working properly.

NOTE

The vast majority of tutorials and resources out there teach you how to build framework-specific components, so keep that in mind when searching for help. If you want to use custom elements, be sure to include that in your search terms.

Earlier in this chapter, we talked about web components. Conceptually, web components and framework components are quite similar. Practically, however, they aren't. At the time I'm writing this, pretty much every JavaScript framework supports standards-based custom elements but also has its own take on and proprietary approach to building components. You'll see this shortly, when we discuss a few popular frameworks.

Using a Framework

Different frameworks have different sets of rules and conventions (we call this being *opinionated*). Some, for example, want you to write *everything* in JavaScript—or, more frequently, TypeScript—files (see the sidebar for an introduction to TypeScript). Yes, including your HTML and CSS. That may throw you for a bit of a loop, as it forces you to adopt a different mental model for building web pages than you've learned in this book.

Frameworks generally have some form of templating system for writing HTML. Some encourage you to write your CSS as JavaScript objects. Others pull your CSS into the framework and spit it out in ways you never imagined, often with all your selectors changed.

Because authoring a website in a framework is so different from traditional web development, and because each framework is so different from the others, it's often quite challenging to move a website off a given JavaScript framework without a wholesale rewrite of the site. Even upgrading from one version of a framework to the next can require a lot of time and effort.

So, knowing how complex JavaScript frameworks are, why would you want to use one? There are two key reasons. First, they give you an incredible amount

TypeScript

If you recall, way back in **Chapter 22, Getting Started with JavaScript**, I introduced the concept of value coercion in JavaScript. If that doesn't ring a bell, perhaps this example will:

```
1 + "1"; // "11"
```

Here, the number 1 was converted (i.e., coerced) into the string value "1" so it could be concatenated with the other string value. JavaScript does this because it is a **loosely typed** programing language. That just means its values are adaptable. The opposite of a loosely typed language is a **strictly typed** one where values are not allowed to adapt to their context. Java is an example of a strictly typed language. You may also hear developers call these "weakly typed" and "strongly typed," respectively.

TypeScript is essentially JavaScript with strict types. At its most basic, TypeScript lets you establish **contracts** in your code. You can think of a code contract like a simple agreement, along the lines of *I expect you to provide me with two arguments that are both numbers and I will return a number to you*.

TypeScript files (which use the *.ts* extension) look a lot like JavaScript files, but they are not interchangeable—they need to be **transpiled** (a portmanteau of "translated" and "compiled") into JavaScript to be executable by a JavaScript engine. At least, that's the case in the browser; all backend JavaScript environments support TypeScript natively.

Using TypeScript can greatly improve the quality of your code and reduce bugs, especially if you're working in a large team. It does this by flagging code issues right in your coding environment (e.g., Visual Studio Code, FIGURE 25-1) and providing details on how to resolve them right then and there.

Here is a simple function that returns the sum of two numbers. As you can see, it looks nearly identical to standard JavaScript, but it contains some extra colons (`:`) followed by the keyword `number`:

```
function add( a: number, b: number ): number {
  return a + b;
}
```

In TypeScript, the colon is used to define the type a variable is expected to have. That could be a standard JavaScript type—like `number`, as shown here—or it could be a custom type you've defined elsewhere in your program.

In the code block above, I've defined the expected types of both arguments to the `add()` function within the parentheses. I've also defined the type the function will return— this is done after the closing parenthesis, but before the opening curly brace. By defining these types, I'm establishing a contract for the function. If I were to try to call `add()` with a string argument, my code editor would let me know I was making a mistake (see FIGURE 25-1). Similarly, if I were to try to use the value returned by `add()` as a string, it would alert me to that issue.

In addition to making it easier for your code editor to highlight bugs, TypeScript can also help it make better code completion suggestions. This is possible because it knows what kind of value you're working with at any given time.

There's a lot more to TypeScript than I can cover here. For complete documentation and to play around with TypeScript, visit *TypeScriptLang.org*.

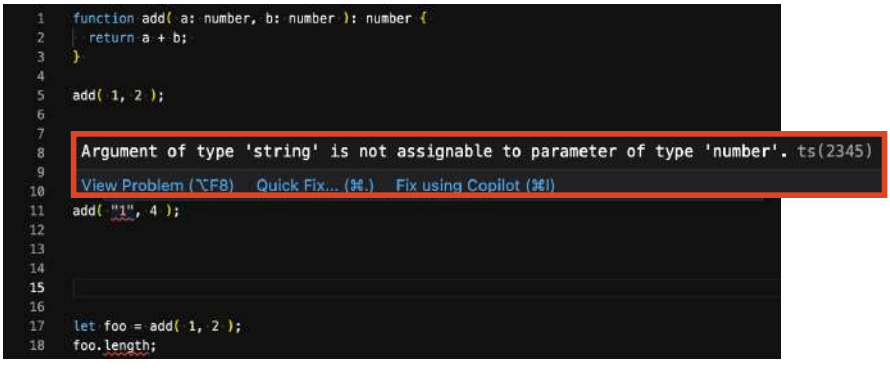


FIGURE 25-1. In Visual Studio Code, the string argument has a red squiggly line under it, indicating there's a problem with my code. Mousing over it, I'm given the explanation that it is expected to be a number and provided with options to help me fix the error. At the bottom, you can also see a red squiggly under the `length` property; this is because number values don't have that property.

of control over how your site is built, deployed, and interacted with. Second, many companies—from small to medium-sized businesses on up to the biggest companies on the planet—use them, so experience with one or more of these frameworks can give you a leg up in the job market.

Pros and Cons of Frameworks

There are pros and cons to using frameworks. We'll start with the positive.

Framework benefits

As I mentioned previously, JavaScript frameworks can help you build complex websites quickly. They do this by providing a ton of building blocks for you. If what you need to build follows common patterns, there is likely a quick and easy way to bring it to life using a JavaScript framework. You don't have to write everything from scratch.

The opinionated nature of frameworks is also a strength because it enforces consistency. That may not seem important when you're a team of 1, but when you're working on a team of 100 or 1,000, it's a big deal. Being able to open a component file and immediately understand what it's doing because it follows the framework's conventions is a real time-saver. That consistency also makes it easier to spot and resolve code errors.

All of the major frameworks also have vibrant communities. Developers help each other solve problems in forums and chatrooms and on collaboration platforms like GitHub. They write and share libraries they've built that add new features to the framework in ways that align with the coding practices of the project. And they write tutorials to help each other get familiar with lesser-known framework features or teach new techniques and approaches for solving common (or less common) problems.

Framework drawbacks

There are plenty of upsides to using frameworks, but they aren't a panacea. First off, not every framework is a good choice for every project. Most JavaScript frameworks were developed for building and managing a specific complex website. For example, React was created by Facebook, Angular came out of Google, and Svelte was created by folks at *The Guardian* newspaper. In all these instances, the frameworks were created to meet the needs of those companies for their specific web projects. They then grew and adapted to the needs of the communities that formed around them. The upshot of this is that unless you're part of a framework's core team, the framework was developed to meet other people's needs. You may have many of the same needs in your project, in which case you'll probably benefit from adopting the framework. Often, however, your project will have different needs, and it can sometimes be hard to adapt a framework to meet them.

If you're building a content-focused website, such as a blog or a recipe sharing or documentation site, using a framework is probably overkill. Because they provide so many features and building blocks for free, JavaScript frameworks generally require that a lot of code be sent to your user's browser. You aren't going to use a fraction of the functionality the framework provides, but you're shipping that code to your users anyway. Generally, for this type of site a framework is not the right tool to use—it's kind of like using a flathead screwdriver to turn a Philips-head screw. The one exception is if you're just looking to build the site to experiment and familiarize yourself with the framework in question.

JavaScript frameworks can be as large as 500 KB, even when compressed—and that's just for the base framework; it doesn't include any of the custom code you wrote for your site. This is orders of magnitude more code than is required to render your typical HTML, CSS, and JavaScript website, which has a base footprint of 0 KB because it only uses the standardized web platform.

Some frameworks are smart(er) about how and when they deliver their massive JavaScript payload to your users' browsers, but all of them create performance challenges. To overcome these, you'll want to make sure you take advantage of server-side rendering (SSR) if the framework provides it. That will at least ensure the HTML for your site gets composed on the server and sent to the browser with the first request so the page can start rendering.

FURTHER READING

For more on the performance implications of JavaScript frameworks, read "The Cost of JavaScript Frameworks" by Tim Kadlec (*oreil.ly/smhUZ*). It was written in 2020, so the numbers are likely a bit off as most of the frameworks he discusses have had numerous releases since then, but its central message holds.

Popular Frameworks

Now that you have a general understanding of what JavaScript frameworks are, I'll discuss a few of the most popular frameworks—React, Angular, Vue, and Svelte—and give you a sense of what it looks like to write code using them. React is currently the most popular JavaScript framework, followed by Angular (see **Note**). Those two can be a bit daunting to learn (and, frankly, look at), however. If you'd like to ease in by learning about a framework that more closely mirrors the approach this book has taken, you might consider reading up on Svelte first, then Vue, before tackling React and Angular.

Consider this a choose your own adventure 😉.

NOTE

Frameworks go in and out of fashion, so others may have surpassed these in popularity by the time you are reading this. Nonetheless, learning how various existing frameworks address components will help you understand the big picture, regardless of which tools are currently in vogue.

Meet React

According to the 2024 State of JavaScript survey (*oreil.ly/hB-Hw*), React is the most popular JavaScript framework out there. React actually calls itself a library, because *technically* it only manages the DOM and doesn't handle more complicated things like navigation and loading data. That said, tools like Create React App allow you to stand up React as a framework, so I feel like the developers are splitting hairs. It's also worth noting that React sits at the center of multiple frameworks, including Next, Remix, and Gatsby.

Promises and Asynchronous Code

All the code we've written in the previous chapters is what we call **synchronous code**. That means it is expected to execute immediately, and each statement in your program is evaluated before the program moves on to the next statement.

This approach is fine for operations that execute quickly, but sometimes the task you need to do will take an indeterminate amount of time to complete. You might need to execute a complex database query that takes a while to run, for example. Or you might need to get the contents of a file on the web, where network conditions and the file's size will impact how long this takes. Using synchronous code for such an operation would make a browser unresponsive until the task was complete. That kind of experience isn't going to make any user happy.

To address this challenge, JavaScript added support for running code **asynchronously**. Asynchronous (or "async") code relies on a special kind of object called a `Promise`. A promise keeps tabs on the long-running operation and allows you to write code that responds to the successful (or unsuccessful) completion of that task. The most important part is that it doesn't hold up the remainder of your code from executing.

As you begin doing more advanced JavaScript work, you'll encounter promises all over the place. Service workers (discussed later in this chapter) make extensive use of them, as do JavaScript frameworks. For more on how they work and some excellent exercises that walk you through the process, check out "What Is a Promise? JavaScript Promises for Beginners" by Kingsley Ubah (*freecodecamp.org/news/what-is-promise-in-javascript-for-beginners/*).

React's major differentiator is that it maintains a copy of the DOM—called the **virtual DOM**—in memory. This allows it to aggregate any changes you make to markup in the virtual DOM and copy them to the actual DOM in one go. This improves performance because (as we discussed in **Chapter 22**) any time you interact with the DOM, whether to read or to write, it has a negative effect on performance. Making all the changes at once minimizes that effect.

As with pretty much every framework, the building blocks you'll work with most frequently in React are components. React components can be created in a few different ways, but perhaps the most straightforward is as a function. Here is a basic React component that creates a paragraph with some standard text and a dynamic value (`name`). It also includes some CSS that sets the text color to `rebeccapurple`:

```
const MyComponent = () => {
  const name = 'Aaron';
  const styles = {
    color: 'rebeccapurple',
  };
  return <p style={ styles }>Hello from { name }</p>;
};
```

Starting at the top:

`const MyComponent = () => { ... }`

This defines the component, `MyComponent`, as a function. Because I used the `const` keyword, the definition of `MyComponent` cannot be changed. If the `() => { ... }` looks strange to you, that's because we haven't talked about it yet. Check out the upcoming **"Arrow Functions"** sidebar for more.

`const name = 'Aaron';`

This is the first statement in the body of the component. It sets "Aaron" as the value of the variable `name`.

`const styles = { ... }`

This is the second statement in the body of the component. It uses a JavaScript object to define the styles that will be applied to the paragraph element and saves them to the variable `styles`.

`return <p style={ styles }>Hello from { name }</p>;`

This is the portion that defines the markup the component will create. The syntax being returned here is called JSX. JSX is an extension to JavaScript that was created by the React team. It needs to be transpiled to normal JavaScript to be usable, so you can't drop this into your code willy-nilly. In this example, it creates a paragraph, assigns inline styles to it (by replacing `{ styles }` with the value of the variable `styles`), and inserts the content "Hello from Aaron" (by replacing `{ name }` with the value of `name`).

You would use the component like this, treating its name as the element name:

```
<MyComponent />
```

This example just scratches the surface of how to build components in React. If you'd like to get a bit more familiar with how they work, check out the React Playground at ***playcode.io/react*** or the W3Schools "React Getting Started" tutorial (***w3schools.com/react/react_getstarted.asp***).

Arrow Functions

The React example introduced you to a new syntax we call arrow functions: `=>`. "Arrow functions" is not the official name of this modern JavaScript feature. The official name is "arrow function expressions," but I've yet to hear a single developer call them that. Arrow functions, despite their name, are not a special kind of function. They are just a more compact way of declaring a function. To illustrate the difference, consider this function declaration:

```
function add( a, b ) {
  return a + b;
}
```

This is how we've been writing functions throughout the last couple of chapters. It's got the **`function`** keyword, the function name, the parentheses with arguments inside, and the function body inside the curly braces with a **`return`** statement.

If I were to rewrite this function using the most compact version of the arrow syntax I can, it would look like this:

```
const add = ( a, b ) => a + b;
```

Looks a little weird, right? You can tell it's an arrow function because of the `=>` bit, which is the arrow. At first blush it's a lot less code, but it's also kind of hard to tell what's happening. I'll break it down into the key points:

- **Variable assignment:** Since JavaScript treats a function like any other value, we can assign a function to a variable, and that's equivalent to naming it using the **`function`** keyword.
- **Arrow:** Think of the arrow as an alternative to the **`function`** keyword. And whereas you put **`function`** to the left of the parentheses, you put `=>` to the right, between the parentheses and the curly braces. Ah, but there are no curly braces here…what gives?
- **Curly braces:** If the body of your function is a single statement, this shorthand allows you to skip the curly braces entirely, as I have here. If you've got more than one statement, you'll need to include the curly braces around the body.
- **Return values:** The final bit of magic here is that when you don't include curly braces, the function will return the result of the statement without you having to explicitly tell it to do so with the **`return`** keyword.

You'll see developers use arrow functions a lot when they need to pass one function to another function as a value—for example, in event listeners. Here's a revised version of an event listener we saw in **Chapter 24**:

```
$button.addEventListener("click", () => console.log("Ouch!"));
```

To be clear, you're under no obligation to use this syntax. I merely wanted to alert you to it because a lot of developers like how compact arrow functions are. They show up quite a bit in JavaScript frameworks as well, and I don't want you to be confused when you see one.

Meet Angular

As I mentioned earlier, Angular is a web framework created by Google. It is the framework of choice for many enterprise-scale websites and includes a ton of code for managing everything from processing forms to how users move from page to page within the site. It even includes its own suite of developer tools and a command-line interface that you can use for running a project locally, testing your code, and more.

To give you a sense of how components are created in Angular, here is a rewrite of the React component I shared earlier:

```
import { Component } from '@angular/core';

@Component({
  selector: 'my-component',
  template: `
    <p>Hello from {{ name }}</p>
  `,
  styles: `
    :host {
      color: rebeccapurple;
    }
  `,
})
export class MyComponent {
  name = "Aaron";
}
```

This is a lot more complicated than the JavaScript we've seen so far, so let's break it down, starting at the top of the code block:

`import { Component } from '@angular/core';`

This **`import`** statement says to add something named **`Component`** from the JavaScript module **`@angular/core`** to the current file.

`@Component({ ... })`

This is what the Angular framework refers to as a **decorator**. In this case, it's telling the framework that the **`MyComponent`** class below is a component. Decorators are not a standard part of JavaScript (although they're currently a proposal), so this is not standard JavaScript that will work outside of Angular. Inside the decorator's parentheses is an object that adds metadata to the component's class that helps Angular create the component.

`selector`

This is the CSS selector that identifies when this component should be inserted. I've given it the string "my-component", which will cause Angular to generate this component when it encounters a **`my-component`** element in HTML:

```
<my-component></my-component>
```

`template`

: This is a template string (i.e., it uses backticks, `) containing the HTML you want Angular to inject whenever it encounters a `my-component` element. In this case, the template string contains a **template property** called `name`. Template properties are easy to spot as they are surrounded by double curly braces (`{{}}`). You might hear this referred to as **mustache syntax**, because those curly braces kinda look like they belong on a certain Italian plumber of video game infamy. You can think of a template property like a variable that stands in for a bit of dynamic content. In this case, it's the `name` value defined in the class (which we'll come to momentarily). This whole template string gets rendered as a paragraph containing the text "Hello from Aaron."

`styles`

: This is another template string that contains the CSS you want to apply to the component.

`export class MyComponent {}`

: This exports the `MyComponent` class so it can be imported by other code, just as we discussed in the **"JavaScript Modules"** sidebar. Within the curly braces, I have set a value (`name`) that will be piped into the template when the component is turned into actual HTML.

Don't feel bad if this is a bit overwhelming. Like I said, JavaScript frameworks have a steep learning curve. If you're interested in learning more about how to work with Angular, I highly recommend the interactive tutorial provided by the Angular team at *angular.dev/tutorials/learn-angular/*.

Meet Vue

Unlike Angular, which requires complete or nearly complete buy-in to its approach to doing things, Vue was designed to make gradual adoption possible. That means you can try it for one specific part of your site and leave everything else alone, or you can go all-in and build the entire site using Vue.

Vue offers a few different ways to build components. For folks who are just starting out, the Single File Component approach (often abbreviated to SFC) is easiest to adopt. SFCs, as the name suggests, bundle all the code for a component into a single file with a *.vue* extension. Within that file, you put your JavaScript, HTML, and CSS, in that order. Here's an SFC for the component we created earlier in both Angular and React:

```
<script setup>
  const name = "Aaron";
</script>

<template>
  <p>Hello from {{ name }}</p>
</template>
```

```
<style scoped>
  p {
    color: rebeccapurple
  }
</style>
```

What's nice about this approach is that it makes a clear distinction between the three different aspects of the component. This means if all you need to do is tweak the CSS or the HTML, you can just work with that language, as opposed to editing CSS as a JavaScript object, for example.

Breaking this down piece by piece:

- The **`script`** element contains the JavaScript for the component. The setup property indicates that the contents should be executed for every instance of the component. Additionally, any variables declared within this element are available to the HTML template.
- The **`template`** element contains the HTML for the component. Like Angular, Vue uses mustache-style curly braces to indicate where variable values should be inserted (like with **`name`** in this example).
- The **`style`** element contains the styles you want to apply when your component is used. If you include the **`scoped`** property, the styles you define will apply only to the contents of the component. If you leave it out, the styles will apply to the whole page.

Of all the frameworks we've discussed so far, Vue is probably the most approachable. You can learn more about Vue by playing around with it in the Vue SFC Playground (***play.vuejs.org***) or completing the Fundamentals course at VueSchool (***oreil.ly/ZUE4c***).

Meet Svelte

Like React, Svelte doesn't pitch itself as a framework. Technically it's a **compiler**, which means it takes the Svelte code you write and converts that into HTML, CSS, and JavaScript that the browser will understand. Svelte does have a full-stack framework called SvelteKit, though, and when folks talk about Svelte, they're often lumping the two together.

Unlike many JavaScript frameworks, Svelte does not build your site in the user's browser. Instead, it has a build step that creates the resources necessary to render the site. The result is better performance in your users' browsers.

Svelte components are written in a single file (like Vue's SFCs), using the *.svelte* extension. Here's that same component again, Svelte-ified:

```
<script>
  const name = "Aaron";
</script>

<p>Hello from { name }</p>
```

```
<style>
  p {
    color: rebeccapurple;
  }
</style>
```

Like Vue's SFCs, Svelte components offer clean separation between JavaScript, HTML, and CSS. However, the syntax is more familiar, using plain **`script`** and **`style`** elements (no attributes) for JavaScript and CSS, respectively. Any other HTML in the component file is treated as the component's markup. Unlike Vue, however, Svelte routes variable values into markup using a single set of curly braces (**`{}`**)—more akin to React—as you can see here with the **`name`** variable.

To learn more about Svelte and try it out for yourself, I highly recommend the tutorial offered by the Svelte team (***v4.svelte.dev/tutorial/basics***).

In addition to all of these frameworks using the same programming language for the frontend, they also use the same language on the server side. That's right: JavaScript! In the next section, I'll give you a quick primer on writing server-side JavaScript using Node.js and Deno.

JAVASCRIPT BEYOND THE BROWSER

JavaScript was originally created to run in the browser. We sometimes call the browser the **client side** or **frontend** of a website. Developers frequently draw a distinction between the client (a.k.a. a user's browser) and the server when talking about where the code is run. Code executing in a browser is on the client side. All other code, including code executing on a server, a developer's computer, etc. is considered **server-side** or **backend** code.

For a long time, all JavaScript code was client side. JavaScript's popularity, however, led to people wanting to use it outside of the browser too. This interest eventually gave rise to new programming environments like Node.js and Deno, which I'll discuss shortly. It's also led to JavaScript being used outside of a web context entirely—you can write JavaScript programs to run right on your own computer, no server required.

You might be wondering what developers are doing with JavaScript if there's no web page to work with. The answer is *lots*. People have written all kinds of software and a ton of developer tooling using JavaScript, including (to give just a few examples):

- Programs that process contact form submissions and send you an email
- Task runners that let you automate away repetitive tasks like image optimization
- Web servers that can be used to bring your content to the web

- Content management systems that enable you to build large content-driven websites more easily
- Testing libraries that make it easy to find and fix errors in your code
- Tools that let you drive a browser programmatically to take screenshots, test your UI, and automate browser-based tasks

The JavaScript ecosystem is massive, and it's continuing to grow. If you want to write a program in JavaScript, chances are there's a way for you to do it, either using the standard JavaScript library or by integrating third-party code. Before we get too far into this topic, however, I want to give you a sense of what's different about writing JavaScript code to run outside of a browser.

Key Differences

Most of the JavaScript code you write as a frontend web developer is directly manipulating the current web page via the DOM, interacting with the browser, or both. Once you're outside of the browser context, however, neither the browser nor the DOM exists. What you get in exchange is the ability to access the filesystem to read, update, and create files and directories on the current device. You can't do that in a browser. You can also write and distribute programs that can be run from the terminal (a.k.a. command line) on virtually any computer. You can even delve deep into networking and write your own server to host websites, handle push notifications, enable videoconferencing, and more.

For all the differences, though, no matter where you're writing JavaScript, at the end of the day it's still JavaScript. Variables, conditionals, loops, functions…they're all still there. If anything, writing JavaScript on the backend is liberating because you can dictate which JavaScript engine (also called a **JavaScript runtime**; see **Note**) your program requires. That would be similar to being able to dictate which browser and browser version all your users have to use to access your site, which isn't feasible on the web.

When you know the specific JavaScript runtime you're using, you will be able to access the complete list of every language feature it supports. In other words, your code will be running in a known environment rather than an unknowable one. You'll never have that kind of certainty on the open web, where people have older browsers, and browsers are all moving at their own pace when it comes to supporting new features.

NOTE

*Although the terms JavaScript engine and JavaScript runtime are often used interchangeably, they actually refer to two different things. The runtime is a complete environment that includes the engine and augments it with additional tools and functionality. For more information, see "What's the Difference Between JavaScript Engines and JavaScript Runtimes?" by Nicholas C. Zakas (**humanwhocodes.com/blog/2024/03/javascript-engines-runtimes/**).*

Meet Node.js

Node.js, commonly referred to as Node, is JavaScript runtime. That means it's a program—much like your browser—capable of interpreting JavaScript code and running it. At the time of writing, Node.js is far and away the most common JavaScript runtime. No matter what platform you use—Windows,

macOS, Linux, Unix—there's a version of Node.js available for you (see *nodejs.org/en/download*).

Node.js has been around since 2009 and uses the same V8 JavaScript engine that's used in Google Chrome. This means that if your JavaScript code runs well in Chrome, there's a good chance it can be run in Node.js as well (apart from the DOM and browser stuff, naturally).

Web developers primarily use Node.js via the command line or terminal in their development environment. You access Node.js's functionality by typing the **node** command followed by the appropriate syntax for whatever you want to do. For example, if I just wanted to see which version of Node.js was installed on my machine, I could type **node -v** on the command line and it would print the version on the next line:

```
% node -v
v20.18.0
```

That's not very interesting, though. Let's run some JavaScript! Imagine I created a file called *hi.js* and gave it the following contents:

```
console.log("Hi from Node!");
```

I could run that JavaScript with Node.js using the command **node ./hi.js**, which tells the computer to pass the *hi.js* file to the **node** application:

```
% node hi.js
Hi from Node!
```

Node.js provides a bunch of JavaScript runtime–specific features as part of its standard library, such as the **fs** library for working with files and directories. It also comes with a tool called **npm** (see sidebar) that makes it easy to integrate other people's JavaScript code into your own code.

Most JavaScript frameworks use Node.js as their default backend, so if that's a direction you plan on going in, it's a good idea to get some familiarity with it. There are plenty of in-depth crash courses out there, but a solid introduction is "How to Get Started with NodeJS—a Handbook for Beginners" by Krish Jaiswal (*oreil.ly/1sGYB*).

Meet Deno

The Deno (pronounced *dee-no*) project was created by the same person who created Node.js, Ryan Dahl, to address the limitations of Node.js. Deno (*deno.com*) allows you to write programs in either JavaScript or TypeScript.

Once installed, Deno can run the same *hi.js* file I created for Node.js in the previous section. You just swap the **node** command for the **deno** command:

```
% deno ./hi.js
Hi from Node!
```

Ha! I probably should have changed the string in the **console.log()** call—although I ran the JavaScript code in the *hi.js* file using Deno, the string

npm

The **Node Package Manager** (ubiquitously known as simply **npm**) consists of two parts: a registry and a command-line interface. The registry, located at *npmjs.com*, is a centralized index of JavaScript written by other developers. This hub makes it easy to discover JavaScript **packages** (a collective term for libraries, frameworks, and helper functions). The npm command-line tool makes it easy to install and uninstall (i.e., *manage*) JavaScript packages within your project's codebase.

If you installed Node.js, you got npm too. You can verify this by typing **npm -v** on the command line to see the version you're currently running:

```
% npm -v
10.8.2
```

Typically, installing a JavaScript package is as simple as typing **npm install** and the name of the package. For example, the following command would install the Vue framework:

```
% npm install vue@latest
```

However, depending on your team's workflow or the specific package's requirements, there may be additional steps you'll need to take.

Incidentally, all the JavaScript frameworks we've discussed are installable with npm. Some come as a single package, and others are divided up into individual packages that each focus on a particular aspect of the framework.

In addition to making it easy to integrate code other developers have written, npm also provides tools to package up your own code and share it via the registry.

inside still says "Hi from Node!" I suppose, if nothing else, this demonstrates that the two runtimes are largely interoperable. That's not surprising, as they both run JavaScript.

Deno differs from Node in a few fundamental ways, though. First, whereas Node provides direct access to the machine your code is running on, Deno's default configuration limits your access to system resources to improve security. So, for example, if you want access to the filesystem, you must explicitly state that your script will use that feature when running it. Similar permissions are needed for network access.

Another difference is that Deno lets you import code directly via URLs. It has its own central repository, JSR (*jsr.io*), but Deno also supports importing Node internals and packages from npm. To use those, you prefix the applicable package name with `node:` or `npm:`, respectively.

At the time of writing, Deno is not nearly as popular as Node.js, but it's frequently used to run **serverless** code. When developers use this term, they're typically talking about code that gets run by a hosting provider in an environment the developer doesn't have to manage. Instead, you provide the code—usually JavaScript—needed to complete a discrete task and upload it into their system.

To illustrate how serverless works, imagine you've written JavaScript to handle submissions to your site's contact form. When someone submits your form, it will send the request to the hosting provider, which will quickly load and run your JavaScript to receive the request and relay the message to you. Once the job is done, the host will spin down the process and wait for the next inbound request.

This approach is quite different from traditional hosting, where you've got a server running all the time, ready to respond to any request. Because it needs to be spun up quickly, it's important that the code run very quickly as well. Deno is much faster than Node.js, which is why it's a common choice.

To learn more, check out the "Getting Started" docs from the Deno team (*docs.deno.com/runtime/*).

Throughout this chapter, my goal has been to open your eyes to the expansive universe of JavaScript and all of the different directions you can take your web development career in using it. Before we wrap things up, I want to touch on one final topic that showcases the awesome power of both JavaScript and the web platform more broadly: progressive web apps.

PROGRESSIVE WEB APPS

The term **progressive web apps** (**PWAs**) was coined to describe a new category of websites that were blurring the lines between experiences that live on the web and traditional software. When you install a PWA, it lives alongside

the other apps you have installed on your device, whether that is a mobile phone, tablet, computer, or something else entirely.

As the name suggests, progressive web apps are a lot like applications in the traditional sense—productivity apps, games, and the like—but they live on the web at a URL. The "progressive" in the name is a nod to progressive enhancement (which we discussed in **Chapter 24**) and reflects the idea that PWAs are traditional websites enhanced to do more. The "more" in this case includes a long and ever-expanding list of capabilities that used to be limited to traditional software. For example, PWAs can:

- Allow a user to work offline. Users can continue to do their work even if their network connection goes down. When the connection comes back, data stored locally can be sent back to the server.
- Show up with other apps. Your website's icon can appear right alongside the other apps your users access daily (e.g., in the list of apps, on the home screen, or in the Windows menu), which can make it easier for them to return to your site.
- Send notifications to a user's device. You can notify users of messages and drive reengagement with your site.
- Store more data on a user's device. Some browsers lift storage restrictions, making it possible for you to allow users to download large, high-resolution videos for offline viewing.
- Access the user's filesystem. With a user's permission, you can read and write files in specific directories on their devices, enabling you to create your own writing, drawing, and coding tools.
- Connect to devices over Bluetooth and USB. If you're working at a company that also produces hardware, you can use JavaScript to communicate with those devices.
- Access a device's camera and microphone. You can enable users to record themselves or have conversations with others.
- Geolocate a user's device with high precision. If you're building a mapping tool or want to automate certain tasks when a user is near a certain location, you can do that in JavaScript.

You get several more features when a user decides to install your site on their device, too (which we'll discuss in the following section). To be clear, you don't have to use all these features to have a PWA. There may only be one or two that make sense for your site, and that's totally cool. The progressive nature of the PWA approach lets you pick and choose, integrating only what you need.

Given the name, you might think these enhancements are only available to something you'd consider to be a web-based application, like Google Docs or Figma. That couldn't be further from the truth. PWAs have been embraced by major companies across the globe, including Airbnb, Facebook, Flipboard,

Flipkart, Forbes, Google, Hulu, Instagram, Microsoft, Pinterest, Spotify, Starbucks, Tinder, Trivago, Uber, *The Washington Post*, The Weather Channel, WhatsApp, YouTube, and plenty of others. If you've got any of these apps on your device, you may be using a PWA and not even know it! Countless smaller-scale developers have also turned their own sites into PWAs. Any website—including yours—can (and probably should) be a PWA.

Core Technologies

At their most basic, PWAs are built on three core technologies:

- HTTPS
- Web app manifests
- Service workers

Let's briefly look at each of these and what its role is in bringing a PWA to life.

HTTPS

As you've seen, PWAs may be granted a whole host of extra privileges in an operating system (e.g., access to geolocation data, the camera, and the filesystem). With that kind of access to potentially sensitive information, it's critical that the connection between your web server and your user's device be secure. To enforce that, many of the features of PWAs are only available to you if your site's URL starts with `https`. Jennifer introduced you to HTTPS and SSL in the sidebar "Moving from HTTPS to Secure by Default" in **Chapter 2, How the Web Works**.

Web app manifest

A **web app manifest** is a file that contains metadata about your website. Jennifer briefly mentioned HTML metadata in **Chapter 4, Creating a Simple Page**. In that introduction, you learned a bit about `meta` tags. A web app manifest is like a collection of `meta` tags, but it differs in two ways:

- It contains metadata about *the whole site*, not a single page.
- The file is written in JavaScript Object Notation (see the upcoming sidebar **"JSON"**) with either a *.json* or a *.webmanifest* extension.

A web app manifest contains information that defines how you'd like your website to show up when it's installed on someone's device. That includes things like the site's name, a short name (for when there's limited space to display the name), icons, the URL that you want to load when it's clicked, and a whole lot more.

You add a web app manifest to your website by referencing it using a `link` tag in the `head` of your HTML:

```
<head>
  <!-- other tags -->
  <link rel="manifest" href="/manifest.json">
</head>
```

Sites that are securely delivered and have a web app manifest become installable. To advertise this, some browsers include an ambient installation badge in the URL bar (see FIGURE 25-2) or promote installation with an overlay at the bottom of the page. In browsers that support this behavior, you can use the **`beforeinstallprompt`** DOM event to programmatically prompt a user to install your site. Other browsers don't promote installation of PWAs, but they do support it via their "Add to Home Screen" (or similar) features.

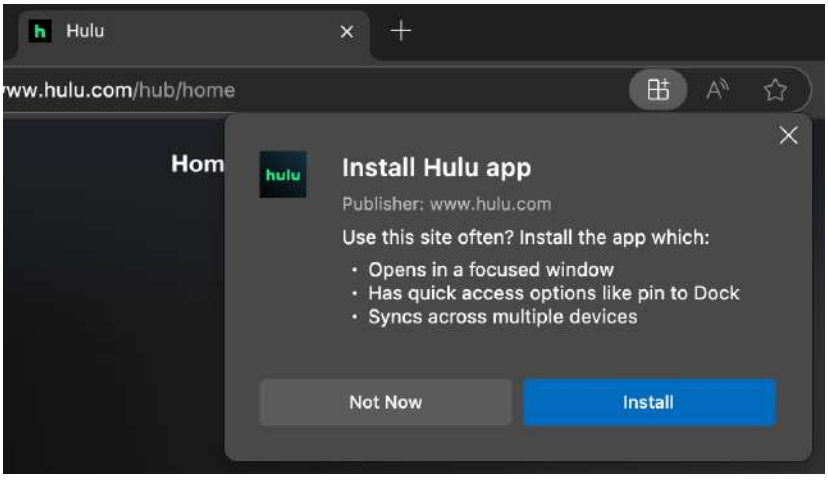


FIGURE 25-2. The install prompt for Hulu in Microsoft Edge on macOS

Once installed, your website's icon will sit alongside the user's other applications (see FIGURE 25-3).

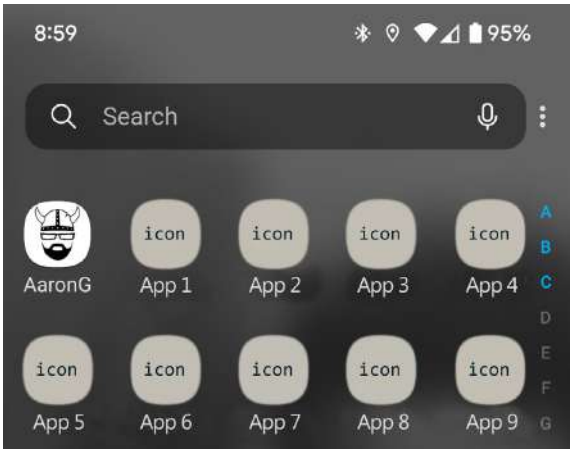


FIGURE 25-3. My website, which is a PWA, installed alongside other apps in my Android phone's apps list

JSON

As JavaScript's popularity has increased, developers have repurposed many of its features. One example of this is JavaScript Object Notation, or JSON. **JSON** (pronounced *jay-sawn*) is a structured way to store information as key-value pairs. This makes it handy for storing structured data locally or moving data to and from your server.

You can embed JSON directly in your JavaScript as a string (with quotes around it), but you can also create JSON files—like the web app manifest—by writing raw JSON in a document (without the outer quotes) and saving it with a *.json* extension.

Here's an example, which is a reduced version of the web app manifest on my site:

```
{
  "name": "Aaron Gustafson",
  "short_name": "AaronG",
  "description": "The online home and work of Aaron Gustafson.",
  "theme_color": "#27831B",
  "background_color": "#fffcf4",
  "icons": [{
    "src": "/i/icon.png",
    "type": "image/png",
    "sizes": "512x512",
    "purpose": "any"
  }]
}
```

Most of these are simple key/value pairs—the name of the site, a description, some colors. As you can see, the keys (to the left of the colon) are all surrounded by quotes, as are the values.

The `icons` key is a bit different, though; its value is an array of JSON objects with their own keys (I've shown only one for simplicity). As this demonstrates, you can store pretty much any kind of value in JSON, making it a super-flexible format.

For more on JSON, start with these articles:

- "JSON for Beginners—JavaScript Object Notation Explained in Plain English" by Tapas Adhikary (*freecodecamp.org/news/what-is-json-a-json-file-example/*)
- "A Beginner's Guide to JSON with Examples" by Atta-Ur-Rehman Shah (*attacomsian.com/blog/what-is-json*).

There are lots of reasons you may want your website to show up as an app. Boosting reengagement with your site is a big one. Another is the opportunity to more directly interact with the user's operating system. Here's a short list of interactions that become possible when you're installed:

Target for sharing

Your site can become an option for the OS's "share" feature, allowing users to directly pass images, links, videos, and the like directly to it. This is super helpful if you're building a social media or image sharing app.

App shortcuts

You can enable quick access to key pages in your site when a user right-clicks or long-presses on your app's icon.

URL handling

Links to your site can open directly in your website's app rather than the browser.

Customized window

When your site is installed, you can use the `display` member of the web app manifest to define how much of the surrounding browser chrome should be included when your site is opened as an app. Features like Window Controls Overlay even allow you to push web content into the app's title bar.

The cool thing is, you can configure all these features and more right from your web app manifest!

Service workers

The last component of progressive web apps I want to touch on is something called a **service worker**. I think of a service worker as a little octopus operating one of those old-timey telephone switchboards. When you create a service worker, it gets installed into the browser from your website. Once installed, it sits in the background and gets access to the browser's internal networking processes, meaning it can observe, redirect, and even respond to requests for just about anything—new web pages, CSS or JavaScript files, images, videos, etc. That may sound dull, but the things you can do with it are anything but.

For instance, when the browser requests a particular URL from your site, you can route that request through your own code and put a copy of the server's response into the browser's cache. Then, if the user requests that same URL again, you can skip the network request and resolve the request with the cached copy you stored earlier. That's a very basic example of how you can use a service worker to make your site work offline.

That's not all you can do with service workers, though. Developers use them to relay push notifications to their users, which is possible because a service worker runs in the background, even when the browser is closed. Developers also use service workers to synchronize data between the user's browser and the server, which enables them to do things like preload the top stories on a news website every morning.

We don't have time to get into the nitty-gritty of writing a service worker, but here's some suggested reading if you'd like to learn more:

- *Going Offline* by Jeremy Keith (self-published; ***goingoffline.adactio.com***)
- The "Service Workers" reference page on web.dev (***web.dev/learn/pwa/service-workers***)

Further Reading

If creating a PWA sounds a bit daunting, it needn't be. The cool thing about progressive web apps is that you can start small and add to them over time. With a good tutorial, you can turn your site into a simple PWA in about 30 minutes. From there, you can develop it further to create even more engaging app-like experiences for your users. If you'd like to learn more about this topic:

- For a good overview of why you might want to build a PWA, read my article "Yes, That Web Project Should Be a PWA" (**oreil.ly/KFXYQ**).
- For a more technical deep dive, I recommend Nitya Narasimhan's series *#30DaysOfPWA* (**dev.to/nitya/series/16849**).

WRAPPING UP

Congratulations, you've made it to the end—not only of this chapter, but of the JavaScript section of this book!

Over the course of these four chapters, you've learned about variables, functions, loops, conditionals, and many more fundamental JavaScript concepts. You've also learned about working with the DOM to alter web pages and enable them to respond to user interactions. Along the way, you've picked up many skills that will help you start your web development career off on the right foot—should you decide that's what you want to do. You've also gotten a peek at what awaits you if you really enjoyed this JavaScript adventure and can't wait to go deeper with a framework or build a progressive web app.

Further Reading

If you're wondering where to go next in your JavaScript journey, here are some great places to start:

- *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly) is a fantastic and deep resource for all things JavaScript.
- *Learning JavaScript Design Patterns* by Addy Osmani (O'Reilly) is an excellent resource for learning about the ways in which developers structure their code and why.
- *JavaScript Enlightenment* by Cody Lindley (O'Reilly) is aimed more at intermediate to advanced JavaScript developers, but it will really help you understand the language and how best to use it.
- *Maintainable JavaScript: Writing Readable Code* by Nicholas Zakas (O'Reilly) is an excellent guide to writing JavaScript with an eye toward your future self, who will have to edit and update the code you write today.

- *Adaptive Web Design: Crafting Rich Experiences with Progressive Enhancement* by Aaron Gustafson (self-published—yeah, shameless plug) discusses JavaScript in the context of web design, with a focus on accessibility, usability, and performance.

Again, congratulations on finishing this section of the book, and thank you so much for reading! Alrighty then, let's dive into this final quiz.

TEST YOURSELF

See how well you absorbed the concepts in this chapter. As always, answers are available in **Appendix A**.

1. Who can create JavaScript libraries?
   a. People working at standards bodies
   b. People working at nonprofits
   c. People working at large companies
   d. Anyone, including me
2. What are the two keywords that are hallmarks of working with JavaScript modules?
   a. `import`
   b. `distribute`
   c. `export`
   d. `consume`
3. What character is required in the name of a custom element built using web components?
   a. `*`
   b. `=`
   c. `-`
   d. `~`
4. JavaScript frameworks typically break a web page's UI into pieces called _____.
5. Can components be nested within one another?
   a. Yes
   b. No

Learn with Others

The following websites can help you connect with other JavaScript developers, get questions answered, and share your own knowledge too:

- **Stack Overflow** (*stackoverflow.com*) is a great place to look for answers to your questions and post new ones. As with most community sites, don't trust that every respondent is right; look for them to cite their sources and look for consensus around answers.
- **FreeCodeCamp** (*freeCodeCamp.com*) provides free training materials and a forum for talking to other learners.
- **Discord** (*discord.com*) has a bunch of JavaScript-focused communities as well as communities dedicated to specific libraries and frameworks.

Local meetups are also a great way to connect with other developers in your area. Most mid-sized cities have a dedicated JavaScript meetup.

6. What do we call the value returned by asynchronous JavaScript?
   a. `Promise`
   b. `Reservation`
   c. `Delay`
   d. `Timeout`
7. Which of the following are considered benefits of using JavaScript frameworks?
   a. Quickly build common types of websites
   b. Slower performance
   c. Coding conventions that enforce consistency
   d. Vibrant communities
8. When you're writing JavaScript to run in Node.js or Deno, what features are not available to you?
   a. Math
   b. The DOM
   c. Promises
   d. Browser features
9. What does PWA stand for?
   a. Progressive web activity
   b. Progressive web app
   c. Performant web app
   d. Portable web activity
10. What is the file format of a web app manifest?
    a. JavaScript
    b. TypeScript
    c. JSON
    d. YAML

V

WEB IMAGES

CHAPTER
26

WEB IMAGE BASICS

Unless you plan to publish text-only sites, chances are you'll need images. For many of you, that might mean getting your hands on an image-editing program for the first time and acquiring some basic production skills. If you are a seasoned designer, you may need to adapt your style and process to make image files that are appropriate for web delivery.

In **Chapter 7, Adding Images**, we covered the basics of embedding images in an HTML document, and in **Chapter 13, Colors and Backgrounds**, you learned how to use images as backgrounds with CSS. This chapter covers the fundamentals of the images themselves. We'll start by reviewing resources for finding or creating images. We'll then spend some time getting to know the file formats appropriate for web pages and how to pick the appropriate format for your type of image. You'll learn how image and screen resolution affect how we design, and you'll get some hands-on experience making transparent images. Finally, the chapter wraps up with a lesson on how to make Favicons (the icons that appear next to page titles on tabs in the browser) and icons for web apps.

IN THIS CHAPTER

Where to get images

Choosing the best image format

Screen resolution and reference pixels

Working with transparency

Favicons and site icons

FURTHER READING

There's a whole lot more to say about images on the web than I can fit here, so I highly recommend that you also read *Image Optimization* by Addy Osmani (Smashing Media AG). It goes into more technical detail about many of the topics I cover in the next two chapters (file formats, optimization, and automation services, to name a few) and also provides valuable information for measuring and improving image performance. It's a must-read if you see image production and management as part of your responsibilities.

IMAGE SOURCES

You need to *have* an image to save an image, so before we jump into the nitty-gritty of file formats, let's look at some ways to get images in the first place. There are many options, from scanning, shooting, or illustrating them yourself to using available stock photos and clip art, using AI image generators, or just hiring someone to create the images for you.

Create Your Own Images

In most cases, the most cost-effective way to generate images for your site is to make your own from scratch. The added bonus is that you know you have full rights to use the images (we'll address copyright in a moment), and they are specific to your organization or business. Designers may generate imagery with cameras, a drawing program, or scanners:

Photographs

You can capture the world around you and pipe it right into an image-editing program. Depending on the type of imagery you're after, you may get sufficient quality with the camera in your phone. Keep in mind that it's always a good idea to create and save high-resolution versions of your images and export smaller copies as needed.

Electronic illustration

If you have illustration skills, you can make your own image in a drawing or photo-editing application. Every designer has their own favorite tools and techniques. For logos and line drawings, starting with a vector drawing program (such as Figma, Adobe Illustrator, Affinity Designer, or Corel Draw) and then saving the image in a web-appropriate format is a good way to go. There are also many cool drawing and painting apps for tablets like the iPad.

Scanning

Scanning is a great way to collect source material. You can scan almost anything, from flat art to small 3D objects. Beware, however, the temptation to scan and use found images. Keep in mind that most images you find are copyright protected and may not be used without permission, even if you modify them considerably. See the **"Scanning Tips"** sidebar for some how-to information.

■ SCANNING TIPS

When you're scanning sources for use on the web, these tips will help you create images with better quality:

- Because it is easier to maintain image quality when you are resizing smaller, as opposed to resizing larger, scan the image larger than you actually need. This gives you more flexibility for creating other sizes later.
- Scan black-and-white images in grayscale (8-bit) mode, not in black-and-white (1-bit, or bitmap) mode. This enables you to make adjustments in the midtone gray areas once you have sized the image to its final dimensions and resolution. If you really want only black and white pixels, convert the image as the last step.
- If you are scanning an image that has been printed, you need to eliminate the dot pattern that results from the printing process. The best way to do this is to apply a slight blur to the image (in Photoshop, use the Gaussian Blur filter), resize the image slightly smaller, and then apply a sharpening filter. This will remove those pesky dots. Make sure you have the rights to use the printed image, too, of course.

Stock Photography and Illustrations

If you aren't confident in your design skills, or you just want a head start with some fresh imagery, there are plenty of collections of ready-made photos, illustrations, buttons, animations, and textures available for sale or for free.

Stock photos and illustrations generally fall into two broad categories: rights-managed and royalty-free.

Rights-managed

Rights-managed means that the copyright holder (or a company representing them) controls who may reproduce the image. In order to use a rights-managed image, you must obtain a license to reproduce it for a particular use and for a particular period of time. One of the advantages to licensing images is that you can arrange to have exclusive rights to an image within a particular medium (such as the web) or a particular business sector (such as the healthcare industry or banking). You also know that the source of the image is verified (i.e., it is not stolen).

On the downside, rights-managed images get quite pricey. Depending on the breadth and length of the license, the price tag may be many thousands of dollars for a single image. If you don't want exclusive rights and you want to use the image only on the web, the cost is more likely to be a few hundred dollars, depending on the source. Getty Images (***gettyimages.com***) is the largest stock image house for rights-managed images, having acquired most of its competitors.

Royalty-free

If you don't have a four- or five-digit budget for images, consider using royalty-free artwork for which you don't need to pay a licensing fee. **Royalty-free** artwork is available for a one-time fee that gives you unlimited use of the image, but you have no control over who else is using it. As of this writing, Getty Images' royalty-free images start at US$175 for a small image appropriate for the web. iStockPhoto (***istockphoto.com***) is easier on the wallet; it has a large collection of images starting at around US$12 a pop.

Creative Commons

Another way to get royalty-free images is to find photos and drawings released by artists under a Creative Commons license. There are a few types of Creative Commons licenses, so be sure to check the terms. Some artists make their work free to use however you want; some artists ask only that you give them credit ("attribution-only"); and some limit the image use to noncommercial purposes.

There are a number of resources for images released under a Creative Commons license, but these are three good first stops:

Unsplash ***(unsplash.com)***

> Unsplash provides free images of top-notch quality, "gifted by the world's most generous community of photographers." It is the source of many of the images in this book.

FURTHER READING

For more information about Creative Commons licenses, go to *creativecommons.org/licenses/*.

Wikimedia Commons *(commons.wikimedia.org)*

A sister site to Wikipedia, Wikimedia Commons is a vast resource of millions of Creative Commons and public domain images and other media files. They are contributed by the community and free to use.

Flickr Creative Commons *(flickr.com/creativecommons)*

The photo-sharing service Flickr has a section dedicated to photos released under a Creative Commons license. The quality varies, but I can usually find what I need for the cost of a photo credit, which is how I found the adorable red panda in FIGURE 26-1 that has been gracing the pages of *Learning Web Design* since its fourth edition in 2012.

FIGURE 26-1. Who doesn't love an adorable red panda? (Photo by Tara Menne.)

Clip Art and Icons

Clip art refers to collections of royalty-free illustrations, animations, buttons, and other doodads that you can copy and paste for a wide range of uses. There are many resources online, and the good news is that some of these sites give graphics away for free, although you may have to suffer through a barrage of pop-up ads. Others charge a membership fee, typically between US$10 and $200 a year. The drawback is that a lot of clip art is poor quality or kind of hokey (but then, "hokey" is in the eye of the beholder). These two sites will get you started, but a web search for "clip art" will turn up plenty more:

Clipart.com *(clipart.com)*

This service charges a membership fee, but it's well organized and tends to provide higher-quality artwork than the free sites.

Freepik *(freepik.com/free-photos-vectors/clip-art)*

Freepik offers a collection of free clip art in addition to its Premium paid service, which provides an impressive collection of illustrations, photos, and icons.

It's also easy to find icons for web pages and applications for free or for a low price (a simple search for "free icons" will do the trick). These image resources specialize in icons:

Google Fonts *(fonts.google.com/icons)*

You can customize and download hundreds of useful UI icons from the Material Design System at the Google Fonts site.

The Noun Project *(thenounproject.com)*

The Noun Project collects and organizes classic one-color icons from around the world. Dozens of collections are available for free in exchange for attribution, and a modest yearly subscription fee gives you access to everything.

Icon Finder *(iconfinder.com)*

This is a good resource for full-color icons of all styles. Some are free, but most are available via a monthly subscription plan. Be sure to check the terms of the Creative Commons license, which varies by icon set.

AI Image Generators

A newfangled way to create an image is to feed a descriptive text prompt into an AI image generator and see what it produces. For example, FIGURE 26-2 shows what OpenAI's Dall·E 3 generated when I typed in "children's book illustration of a bulldog riding a Harley motorcycle through Ohio at sunset."

FIGURE 26-2. An image generated using Dall·E 3 by OpenAI

AI image generation went mainstream in 2021 with the release of Dall·E. Since then, there has been an explosion of new tools, and AI has made its way into design tools by Adobe, Figma, and others. Although convenient (and a bit magical), using AI-generated images is fraught with ethical and copyright issues because many of these tools are trained on real artists' work, without permission, credit, or compensation. That said, I believe they're here to stay, and hopefully the sticky issues will be smoothed out.

If you'd like to try it yourself, these services will get you started, but there are many more:

- Dall·E 3 by OpenAI (***labs.openai.com***)
- Midjourney (***midjourney.com***)

AI for Code

AI tools like ChatGPT (***chatgpt.com***) have become popular assistants for writing code, including the code for creating websites. I won't be mentioning AI as a generation tool any further in this book because you're here to learn how to do this stuff yourself. But once you develop solid skills, you may find an AI code generating tool to be helpful for efficiency.

Hire a Designer

Finding and creating custom images takes time and particular talents. If you have more money than either of those things, consider hiring a graphic designer, photographer, or illustrator to generate the imagery for your site for you. The advantage of hiring a professional is that you get custom images tailored to your message or brand, not just generic stock images. If you start with high-quality original images, you can use the skills you learn in this book to produce web versions as you need them.

MEET THE IMAGE FORMATS

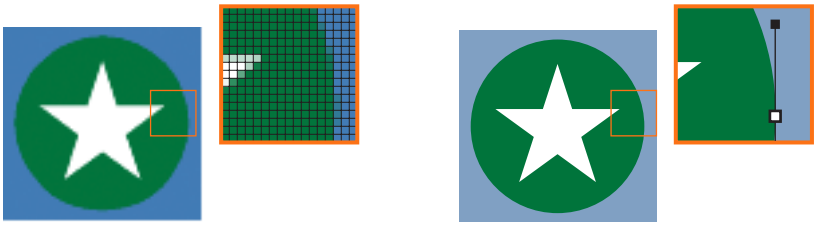
Once you have your hands on some images, you need to get them into a format that will work on a web page. There are dozens of image file formats out there, tailored to specific purposes. On the web, the following formats have the best browser support and are the easiest to produce using standard image editing software:

- JPEG (*"JAY-peg"*)
- PNG (*"ping" or "p-en-gee"*)
- WebP (some say *"weppy,"* but "web-P" sounds just fine to me)
- GIF (*"giff"* or *"jiff"*—it's a hot debate)

These aren't the only formats in town, though. We'll also take a quick look at these state-of-the-art options designed to replace the 30+-year-old JPEG:

- AVIF
- JPEG-XL

What all of these formats have in common is that they are used for **bitmap** (also called **raster**) images made up of a grid of pixels. When you zoom in on a bitmap image, you can see that it is like a mosaic made up of many pixels (tiny, single-colored squares), as shown in FIGURE 26-3, left.



Bitmapped images are made up of a grid of variously colored pixels, like a mosaic.

Vector images use mathematical equations to define shapes.

FIGURE 26-3. Bitmap and vector images

Another extremely popular web image format, SVG (Scalable Vector Graphics), is *not* included in the previous list because it's a bit of an oddball. SVG is a vector image format (made up of lines and shapes defined mathematically; see FIGURE 26-3, right) in which the shapes, fills, filters, and more are generated by XML text instructions. It is so unique, in fact, that I've given it its own chapter: **Chapter 28, SVG**. This chapter focuses on the bitmap formats.

Remember that your responsibility as a web designer or developer is to make sure your pages display as quickly as possible. To limit the amount of data that needs to be downloaded, you should choose the image format that results in in the best image quality at the smallest file size.

You should choose the format that results in the best image quality at the smallest file size.

Naming and Identifying Image Types

It's important to name your image files with the file extension that matches the file type. On occasion, you may need to identify the file (MIME) type of the image format, such as when providing fallback formats with the `picture` element. The MIME type and file extensions for the image formats covered in this book are listed in TABLE 26-1.

TABLE 26-1. File extensions and MIME types for common web image formats

Format	Extension	File (MIME) type
JPEG	*.jpg* or *.jpeg*	`image/jpeg`
PNG	*.png*	`image/png`
WebP	*.webp*	`image/webp`
GIF	*.gif*	`image/gif`
AVIF	*.avif*	`image/avif`
JPEG-XL	*.jxl*	`image/jxl`
APNG	*.apng*	`image/apng`
SVG	*.svg*	`image/svg+xml`

Each format has its own specialties when it comes to image optimization and features. Some work best for photographic images, while others excel at compressing flat colors. Some offer transparency; a few can be used for simple animations. In this section, I'm going to dig *deep* into what makes each web image format tick. It may feel like a lot of technical information, but understanding how the different formats work will help you make the best choice. Along the way, you'll learn terminology that will serve you well when creating, saving, and optimizing images, so pay close attention to these concepts:

- Lossy versus lossless compression
- Progressive display
- 24-bit (full color) versus indexed color models
- Alpha versus binary transparency

If you lack time or patience for all the technical details, you can read the **"Image Format Selection (Spoiler Alert)"** sidebar for a quick summary. Otherwise, let's get started with our stroll through the formats, beginning with JPEG.

Image Format Selection (Spoiler Alert)

If you're in a hurry to find the best format options for particular image types, you can refer to these general guidelines. Although I've spoiled the surprise here, I do recommend you take the time to learn more about each format to aid in your image-making endeavors down the line.

The short story is that the overwhelming majority of images on the web are saved in PNG or JPEG, with WebP coming in a distant third. WebP is supported by all current browsers, but if you need to support very old browser versions or Internet Explorer, provide a fallback image (I'll review how at the end of this section). GIF, once the dominant format, is falling out of use because 8-bit PNG and WebP offer smaller file sizes.

Here is the general format selection strategy:

- For photographs, save the image as a JPEG or a lossy WebP (which will be smaller but may require a fallback if you need to support older browsers).
- Save images with flat areas of color and fine detail as PNGs. An indexed color 8-bit PNG will usually result in the smallest file size.
- If your image has transparent areas, use PNG-24 or WebP.
- If you require animation, use WebP or GIF.

The other formats I mentioned earlier, AVIF and JPEG-XL, are best for photos and can compress image data much more efficiently than JPEG, resulting in smaller file sizes for the same image. For now, they have limited browser support, so they must be used with JPEG or WebP fallbacks.

The Photogenic JPEG

One of the most popular graphic formats on the web is **JPEG**, which stands for **Joint Photographic Experts Group**, the standards body that created it in 1992.

JPEG is an appropriate format to use if your image is a photograph or contains soft, smooth color transitions (FIGURE 26-4). The JPEG compression scheme *loves* gradients and blended colors, but it doesn't work especially well on flat colors (which get blotchy) or hard edges (which get fuzzy) when set at higher compression settings. Let's look at the key characteristics of this format.

JPEG is the best format to use if your image is a photograph or contains soft, smooth color transitions.

FIGURE 26-4. The JPEG format is ideal for photographs (color or grayscale) or any image with subtle color gradations.

24-bit color

JPEG is capable of displaying millions of colors in the RGB color space (RGB color is explained in **Chapter 13**). It's a **24-bit color** (also referred to as **Truecolor**) format because each of the three color channels—red, green, and blue—is defined with 8 bits of information. Displaying 24-bit color is one aspect that makes JPEG ideal for photographs.

Lossy compression

The JPEG compression scheme is **lossy**, which means that some of the image information is thrown out in the compression process (see **Warning**).

WARNING

Cumulative Image Quality Loss

Be aware that once image quality is lost in JPEG compression, you can never get it back again. For this reason, you should avoid resaving a JPEG as a JPEG. The image loss is cumulative—in other words, you lose image quality every time you open and resave the file.

It's better to hang on to the original image and export JPEG copies as needed. That way, if you need to make a change, you can go back to the original and do a fresh save or export.

Fortunately, this loss is not discernible for most images at most compression levels. At high levels of JPEG compression, however, you may begin to see color blotches and squares (referred to as **artifacts**) that result from the way the compression scheme samples the image (FIGURE 26-5).

Original Maximum compression

FIGURE 26-5. JPEG compression discards image detail to achieve smaller file sizes. At high compression rates, image quality suffers, as shown in the image on the right.

In most image editing programs, you can control how aggressively you want the image to be compressed with a quality setting when saving or exporting. This involves a trade-off between file size and image quality. The more you compress the image (for a smaller file size), the more the image quality suffers. Conversely, when you maximize quality, you also end up with larger files. The best compression level to use depends on the particular image and its purpose on the site. I will be covering JPEG compression in more detail in **Chapter 27, Image Production Techniques**.

The best JPEG compression level to use depends on the particular image and its purpose on the site.

Progressive display

Progressive JPEGs display in a series of passes, starting with a low-resolution version that gets clearer with each pass, as shown in FIGURE 26-6.

FIGURE 26-6. Progressive JPEGs render in a series of passes.

The advantage of using progressive JPEGs is that viewers can get an idea of what the image shows before it downloads completely. Also, making a JPEG progressive usually, although not always, reduces its file size slightly. The disadvantage is that progressive JPEGs take a bit more processing power to display, especially on mobile devices. Despite that minor hitch, the best practice is to make larger JPEG images progressive, not only for the smaller file size, but because they appear on the page faster, improving perceived performance. For small images, progressive encoding doesn't offer a significant advantage.

The Wide World of JPEG

The Joint Photographic Experts Group has been busy since releasing the original JPEG format we know and love. It has released several newer JPEG standards (JPEG-XL, JPEG 2000, JPEG XR, JPEG-LS, JPEG XS, and others) that aim to keep pace with changing image requirements in all arenas, from digital cameras to medical imaging. We'll talk about JPEG-XL later in this chapter. Newer formats include features such as lossless compression, the ability to store 16-bit information in the RGB color channels, support for the CMYK color model, and lightweight implementations that are easier to encode and decode. Read more about them at *JPEG.org*.

The Powerful PNG

The **PNG** (**Portable Network Graphic**) format was designed to replace GIF for online purposes and TIFF for image storage and printing. It can be used to save 24- and 48-bit RGB color, 16-bit grayscale, and 8-bit indexed color images, but for the purposes of web production, you need to choose only between 24-bit (**PNG-24**) and 8-bit (**PNG-8**). I'll explain the difference in this section.

Despite getting off to a slow start in terms of browser support, PNG is now the most popular format, and for good reason. It offers an impressive lineup of features:

- 24-bit *or* indexed color formats
- Lossless compression
- Progressive display in multiple passes
- Multiple levels of transparency (alpha) *or* simple on/off transparency
- Embedded gamma (brightness) adjustment information
- Embedded text for attaching information about the image

24-bit color (PNG-24)

A PNG can be saved as a 24-bit image, with each channel (red, green, and blue) defined by 8 bits of information. In many image editing programs, 24-bit RGB PNGs are identified as PNG-24. Mentions of simply "PNG" in software and articles are nearly always referring to this full-color, 24-bit version of the format.

The two key features of PNG-24s are that they are "lossless" and they can contain multiple levels of transparency.

The PNG format can also store 16 bits of information per channel, resulting in 48-bit RGB images. While great for storage of high-quality originals, 48-bit PNGs are not appropriate for the web because of the large file size (for that matter, even 24-bit PNGs may not be the best choice due to file size).

The two key characteristics of PNG-24s are that they are "lossless" and they can contain multiple levels of transparency. Let's take a closer look at these features.

Lossless compression

Unlike JPEG, which uses a lossy compression algorithm where image data is tossed out in order to reduce the size of the file, PNG-24 is **lossless**, meaning no data is sacrificed (see **Note**). Because it is a lossless format, a 24-bit PNG is nearly always significantly larger than a lossy JPEG or WebP of the same image. For that reason, JPEG and WebP (as well as the newer formats I mentioned earlier) are better choices than PNG for photos on the web.

However, PNG-24 was the first format to include a killer feature that has made it the web's most popular format, and that is...

NOTE

If you are burning with curiosity about the technical details of PNG compression, I recommend the article "How PNG Works" by Colt McAnlis (oreil.ly/lNOx7).

Alpha transparency

Although all bitmap images are rectangular by nature, transparency creates the illusion that the image has a more interesting shape. PNG-24 files can contain multiple levels of transparency that allow the background to show through. They do this by storing 8-bit transparency information (256 levels) in a fourth channel, called the **alpha channel** (thus the term **alpha transparency**).

NOTE

You sometimes see PNG-24 with alpha transparency referred to as 32-bit PNG because there are 8 bits for each of four channels: red, green, blue, and alpha.

FIGURE 26-7 shows the same PNG against two different background images. The orange circle is entirely opaque, but the black drop shadow contains multiple levels of transparency (ranging from nearly opaque to entirely transparent), allowing the shadow to blend seamlessly into any background. Alpha transparency will be addressed in more detail in **"Working with Transparency"** later in this chapter.

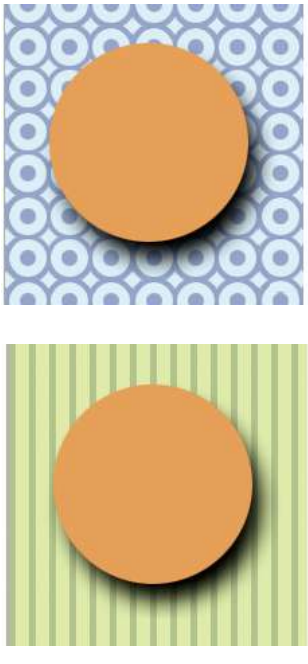


FIGURE 26-7. Alpha channel transparency allows for smooth blending with the background, as shown in the drop shadow around the orange circle PNG.

8-bit indexed color (PNG-8)

The PNG format can also save 8-bit indexed color images, commonly referred to as PNG-8 images, making it a good replacement for GIF. Instead of storing information in RGB color channels, an indexed color image uses a palette of up to 256 colors (more on that in a moment). Because of the limited number of colors, PNG-8 is good for images with flat colors like logos, charts, screenshots, and icons, such as the examples in FIGURE 26-8. You can save photographs or textured images in PNG-8 format too, but they won't be saved efficiently, resulting in larger file sizes. However, PNG-8 does work fine for images with a combination of small amounts of photographic imagery and flat areas of color.

The key characteristics of PNG-8s are that they use an 8-bit indexed color model and they support transparency. Let's explore what that means.

How indexed color works

I mentioned earlier that PNG-8 files are indexed color images that contain 8-bit color information (they can also be saved at lower bit depths). Let's decipher that statement. First, **8-bit** means PNG-8s can contain up to 256 colors—the maximum number that 8 bits of information can define ($2^8 = 256$).

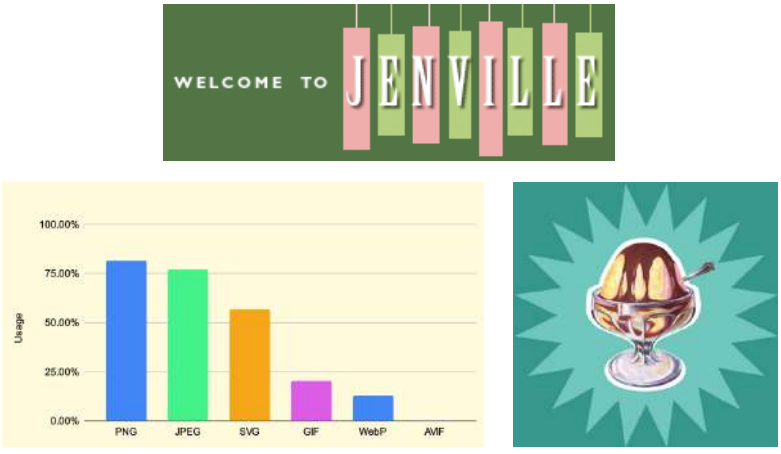


FIGURE 26-8. The PNG-8 format is efficient at compressing graphical images comprising mainly flat colors and hard edges (see **Note**).

NOTE

Because they are simple illustrations, the images in FIGURE 26-8 *could also have been drawn with vectors and saved in SVG format.*

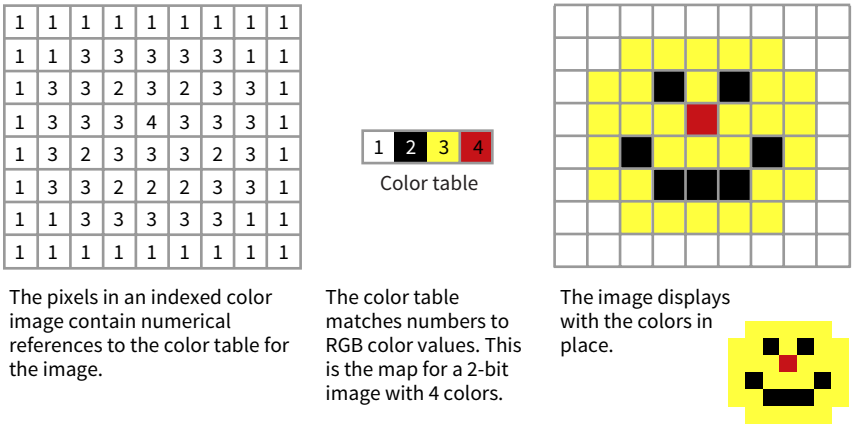
Lower bit depths result in fewer colors and also reduce file size. For example, 4-bit images contain only 16 colors ($2^4 = 16$).

Indexed color means that the set of colors in the image—its **palette**—is stored in a **color table** (also called a **color map**). Each pixel in the image contains a numeric reference, or index, to a position in the color table. Let's make this clear with a simple demonstration. FIGURE 26-9 shows how a 2-bit (4-color) indexed color image references its color table for display. For 8-bit images, there are 256 slots in the color table.

NOTE

GIFs are also 8-bit indexed color images, so this discussion of bit depth and palettes applies to GIFs as well.

Image editing programs generally allow you to view the color table for an indexed color image. In Photoshop, you can view (and even edit) the color table by selecting Image → Mode → Color Table (FIGURE 26-10).



The pixels in an indexed color image contain numerical references to the color table for the image.

The color table matches numbers to RGB color values. This is the map for a 2-bit image with 4 colors.

The image displays with the colors in place.

FIGURE 26-9. A 2-bit image and its color table

Image with 64 indexed colors

Photoshop's Color Table

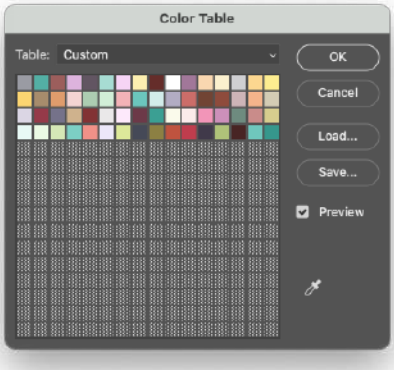


FIGURE 26-10. The color table displays the 64 colors used in the image.

Repetitive compression scheme

PNG's lossless compression scheme takes advantage of repetition in pixel colors. When it encounters a string of pixels of identical color, it compresses them into one data description (FIGURE 26-11). This is why images with large areas of flat color condense better than images with textures and variations in pixel colors.

Most source images (scans, illustrations, photos, etc.) start out in RGB format, so they need to be converted to indexed color in order to be saved as a PNG-8 or GIF. When an image goes from RGB to indexed mode, the colors in the image are reduced to a palette of 256 colors or fewer, a process known as **quantization**. For most programs, including Photoshop, the conversion takes place when you save or export the image. Some image editing programs, like GIMP, may require you to convert the image to indexed color manually first and then export the PNG-8 as a second step.

PNG-8 and GIF compression store repetitive pixel colors as a single description.

"14 blue"

In an image with gradations of color, information must be saved for every pixel in the row. The longer description means a larger file size.

"1 blue, 1 aqua, 2 light aqua..." (and so on)

FIGURE 26-11. A simplified demonstration of a repetitive compression scheme

Transparency

You can make parts of a PNG-8 image transparent so that the background image or color shows through. In the most commonly supported type of PNG-8 transparency, which is known as **binary transparency**, **index transparency**, or **1-bit transparency**, pixels are either fully transparent or fully opaque. This is handled by assigning a fully transparent value to one spot in the color table. We'll look at binary transparency later in this chapter.

Additional PNG features

There are a few other features that make PNG a Pretty Nifty Graphic format (see what I did there?):

Gamma correction

Gamma refers to the brightness setting of a monitor. PNGs can be tagged with information regarding the gamma setting of the environment in

which they were created, allowing them to retain their intended brightness and color intensity in browsers that perform gamma correction. Unfortunately, this feature is not consistently supported.

Embedded color profile information

The PNG format can store the ICC color profile information of the system it was created on. In fact, if you are finding that it is difficult to match an RGB value in a PNG to the same RGB value in a background color, the embedded color profile is likely to blame. The block of code for storing ICC profiles generally gets removed by image optimizers.

Embedded text

PNGs also have the ability to store strings of text. This is useful for permanently attaching text to an image, such as copyright information or a description of what is in the image.

Progressive display (interlacing)

PNGs can be coded for progressive display (like JPEGs), revealing the image in a series of seven passes that gradually fill it in both horizontally and vertically. Interlacing adds to the file size and is usually not necessary; in my experience, most image tools don't give you the option to turn it on anyway.

Animation

The **APNG** (**Animated PNG**) format is an extension to PNG that adds the ability to animate frames. In addition to 8-bit animations, it includes support for animated 24-bit images with alpha transparency. Despite being supported by all modern browsers as of 2019, APNG has never really caught on and is used by only 0.1% of sites as of this writing.

PNG Wrap-Up

Before we move on, here's the skinny on what you should know about PNG:

- If you have a bitmap image with areas of flat color, with or without transparent areas, PNG-8 may offer the smallest file size.
- If you have a photographic image with no transparency, you *could* use PNG-24, but it will always result in a larger file than WebP, JPEG, and other formats with lossy compressions schemes.
- If you need variable levels of transparency regardless of the image type, PNG-24 is the only option with universal browser support, but the file will likely be larger than a lossy WebP of the same image.

Let's take a break here so you can try saving images in JPEG and PNG format in **EXERCISE 26-1** and see how format and image quality affect the file size. Take a moment to read the **"A Word About Image Editing Tools"** sidebar before you get started. After the exercise, we'll move on to WebP.

A Word About Image Editing Tools

There are many programs out there for creating and editing images. I will be primarily using Adobe Photoshop because (for now) it is the go-to tool for professional web designers and it offers a free trial version if you'd like to work along. Unfortunately, I won't be able to teach you how to use Photoshop, but I will demonstrate the features relevant to our lessons, and there are helpful tutorials available at ***oreil.ly/WLmyM***.

You can use the image editing tool of your choice for the exercises, as long as it has the ability to save to JPEG, PNG, and WebP with custom quality and compression settings. Here's a little roundup of a few options that work well for web graphics:

Adobe Photoshop (Mac and Windows)

Photoshop remains the most popular image editing program for design professionals. The major downside is that it is costly and available only via a monthly subscription, so once you get in, you can never get out. If you'd like to give it a spin, a free seven-day trial version is available at ***oreil.ly/Hx0w6***.

Figma (Mac and Windows)

Figma is a vector-based tool for designing and prototyping site and application interfaces. It allows you to export image assets right from your design (see the **"Exporting Assets in Figma"** sidebar after the exercise). The starter version is available for free at ***figma.com***, and Figma offers training for beginners and advanced users alike in its video series at ***www.youtube.com/@Figma***. The Figma Learn site (***help.figma.com***) also provides excellent guides for getting started.

Affinity Photo and Affinity Designer (Mac and Windows)

The Affinity suite of design tools is an affordable alternative to Adobe products if a monthly subscription isn't your cup of tea. Affinity Photo (***affinity.serif.com***) offers an impressive set of image export settings.

GIMP (all platforms)

If you're looking for a free image editor, GIMP (GNU Image Manipulation Program) has many of the same features as Photoshop, and it is absolutely *free*! Like, forever—not just a limited trial. Download it at ***gimp.org***.

Pixelmator Pro (Mac only)

Pixelmator Pro is a solid image tool for the money (***pixelmator.com***).

PaintShop Pro (Windows only)

Corel's PaintShop Pro is an old standby for creating and editing images on Windows (***paintshoppro.com***).

Command-line image tools

Many image conversion and optimization tools are available only via the command line, which means you use them by typing specific instructions after a prompt in a **command-line terminal** rather than interacting with a **graphical user interface (GUI)**. That may seem old-fashioned, but it's much easier for programmers to write and distribute utilities without needing to create a rich GUI layer, so the command-line terminal is likely here to stay.

When it comes to image production, there are handy tools like `cwebp` for converting PNG and JPEG images to WebP format and the corresponding `dwebp` for converting WebP to PNG, as well as various tools for image optimization and other image production tasks.

Command-line tools are central to a web developer's workflow for a whole array of tasks well beyond image manipulation. Mastering the command line is beyond the scope of this book, but I've written an article, **"A Brief Introduction to the Command Line,"** to get you started. It's available online at ***learningwebdesign.com/articles.***

EXERCISE 26-1. Comparing JPEG and PNG formats

In this exercise, we're going to see the effect that image format has on file size by exporting two images in JPEG and PNG formats with various settings, then comparing the file sizes. I've provided two image files with the materials for this chapter (FIGURE 26-12), but you can also experiment with your own. If you are new to image editors, this will give you a feel for exporting images as well.

croissants.jpg

ad-banner.png

FIGURE 26-12. Export images in various formats to see how format affects file size.

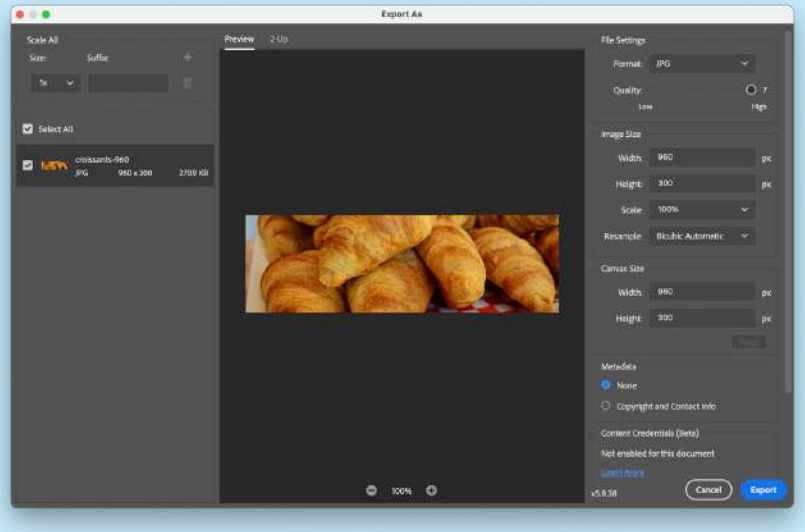


FIGURE 26-13. The Export As panel in Adobe Photoshop 2025 has options for selecting a format, adjusting settings, viewing a preview, and monitoring the resulting file size.

1. First, open *croissants.jpg* in the image editor of your choice and export it in JPEG format. If your image editor does not have an export function, you may need to do a Save As. Be sure that you always start with the original image throughout this exercise.

   If you are using Photoshop, select **File → Export → Export As** (see **Note**). A panel will open that allows you to select the file format and various related settings (FIGURE 26-13). You can adjust the quality with a slider on a scale from 1 to 7 (which is a little odd because most tools indicate quality on a scale from 0–100%). As you adjust the setting, an image preview shows the resulting image and the file size is indicated in the left column. Try different quality settings and write down the resulting file sizes at full quality (7, or 100%), medium quality (4, or 50%), and lowest quality (1, or 10%). Zoom into the preview and pay attention to the quality in the soft areas of the image.

   If you do not have a preview in your tool, you can export three separate JPEGs with the quality set to 100%, 50%, and 10%. Open the images in your image editing tool to check the quality, and use the Finder or File Explorer to check the file size.

2. Now, export the full-color image again and select PNG-24 (in Photoshop, uncheck the "Transparency" and "Smaller File (8-bit)" checkboxes).

NOTE

Photoshop also has a "Save for Web" feature with additional optimization settings, but I'm sticking with Export As in this exercise as it is Adobe's preferred method for exporting images. We'll take a brief look at "Save for Web" when we talk about transparency later in this chapter. For flattened images, you can also choose Save As and select an image format, but you won't get any of the optimization that you do when you export.

EXERCISE 26-1. Continued

3. Finally, export the image as an indexed PNG-8. In Photoshop, check "Smaller File (8-bit)" and the image will be converted from RGB to indexed color on export. In other tools, you may need to convert the image to indexed color first before you export. Choose a palette of 256 colors, if you are given a choice. Be sure to revert the image to RGB color or close it without saving after exporting.
4. Let's see what we've got! Here are the resulting file sizes that I got using Photoshop's Export As feature. Note that if you used another image editing tool, your file sizes are likely to be different because tools use different compression algorithms. They should still be in the same general ballpark, however.

	Original	JPEG (q7)	JPEG (q4)	JPEG (q1)	PNG-24	PNG-8
croissants.jpg	299 KB	277 KB	**42 KB**	12 KB	492 KB	224 KB

Conclusion: In my opinion, the JPEG with a medium setting has the best balance of image quality and file size. The file lowest-quality JPEG is tiny, but if you zoom into the preview or saved file, you will find that the quality is too blotchy. The PNG files are *much* larger because they are lossless.

Of course, quality is subjective, so where you find the best balance will depend on your preferences and the purpose of the image. You might decide that an image is so important that pristine quality is worth the extra download time, but generally, you can shave a lot of bytes off an image without noticeable degradation.

5. Now we're going to repeat all the previous steps, this time using the *ad-banner.png* image. Export the original image as a JPEG at various settings, a PNG-24, and a PNG-8. If you are using Photoshop or another tool with a file size preview, you can simply note down the file sizes.

   These are the results I got—but note that *they don't tell the whole story:*

	Original	JPEG (q7)	JPEG (q4)	JPEG (q1)	PNG-24	PNG-8
ad-banner.png	114 KB	48 KB	11 KB	5 KB	43 KB	**19 KB**

Clearly, the high-quality JPEG and PNG-24 are not the best choices because the file sizes are larger than they need to be. The lower-quality JPEGs have the smallest file sizes, but if you zoom into the previews or exported files, you'll see that the lossy JPEG compression has garbled the fine typography (FIGURE 26-14). The indexed PNG-8 comes in at 19 KB (a good file size savings from the 114 KB original. In **Chapter 27**, you'll see how reducing the number of colors can reduce the file size even further.

Conclusion: For the *ad-banner.png* image, PNG-8 is the winner. It offers the smallest file size while keeping the flat colors artifact-free and the typography crisp.

Original

PNG-8 (256 colors)

JPEG (low quality)

FIGURE 26-14. The typography gets blurry when saving the image as a JPEG with a low quality setting (1). The file size savings is not worth the loss of quality.

The Performant WebP

That takes care of the universally supported JPEG and PNG. You've picked up some useful terminology along the way that will serve you well as we look at the newer WebP format.

Google calls its open source WebP (Web Picture) format "the Swiss Army knife of image formats," and for good reason. Designed specifically for the web, it has virtually all the features JPEG and PNG provide and generally results in smaller file sizes for the same image. There's a WebP 2 format on the horizon as well (see the **"WebP 2"** sidebar).

Here is an overview of what WebP has to offer:

Lossless or lossy compression

WebP can be saved in a lossy format (like JPEG) or a lossless format (like PNG). Its lossy compression scheme uses the same encoding used in the VP8 video codec. It uses "prediction coding" to more accurately represent the image after compression, resulting in better image quality at smaller sizes. An image saved as lossy WebP will likely be 25–35% smaller than the same image saved as a JPEG with comparable quality. Similarly, a lossless WebP will be approximately 25% smaller than a lossless PNG-24.

Alpha transparency

WebP has an alpha channel for multiple levels of transparency. Alpha transparency can be used with either lossless (PNG-like) image compression or lossy (JPEG-like) compression, with the latter resulting in a file that is 60–70% smaller than a transparent PNG-24 of the same image.

Animation

It is possible to animate WebP images. (Sorry GIF, there goes your last remaining advantage.) Search online for WebP animation tools and GIF converters if you'd like to give it a try.

Metadata

Like PNG, the WebP container can store metadata right in its code.

Color profile

The WebP container can embed color profile (ICC) information. However, WebP works in the RGB color space only and can't be used for high dynamic range (HDR) images with wide color gamuts.

WebP 2

I know you're just getting introduced to WebP, but you should know that Google is already working on upgrades with an experimental WebP 2 codec. The new format aims to reduce file sizes even further, in addition to making improvements to transparency, lossless compression, and image quality at the more aggressive compression levels, among other features. If you'd like to follow its progress, go to *oreil.ly/NcW3F*.

Exporting Image Assets in Figma

Figma is currently the most popular tool for designing sites and apps, so you may find yourself working on a team that uses it for collaborative design and development. In Figma, you can export the images and vector illustrations right from the elements in your design. To export an image asset:

1. Select the frame that contains the image.
2. Scroll down to the Export settings in the settings column on the right.
3. Select the file format (JPG, PNG, SVG, or PDF).
4. Click Export Frame.

Figma doesn't provide options for selecting quality or other file size optimizations, so you get what you get. In my experience, JPEGs exported directly from Figma are larger than they need to be—for example, the croissants image from EXERCISE 26-1 came in at 381 KB—but JPEGs and PNGs can be made smaller after export with third-party optimization tools.

But I'm not mad at it! Figma is a vector-based UI design tool, not a bitmap image editor, so there's no reason to expect it to do everything. It does offer the convenience of exporting images at the correct dimensions for your layout, as well as the ability to export multiple sizes for responsive image sets. You may also find plug-ins for image optimization. And because it is vector-based, it's great for saving page elements in SVG format.

When to use WebP

The general consensus is to use WebP when speed is more important than image quality. That's because it throws out more image data than JPEG by default in its lossy compression schemes.

An image saved as a lossy WebP will likely be smaller than the same image saved as a JPEG with comparable quality.

It's a good JPEG competitor at compression settings between 50 and 75%, but at settings below that, the image quality suffers. At higher settings, the file sizes are comparable. If you have high-quality JPEGs, converting them to medium-high-quality WebP images will certainly be worth the effort in terms of data savings. However, if you are starting with a low-quality JPEG with artifacts, converting to WebP will not fix them.

WebP's ability to combine alpha transparency with lossy compression in the color channels gives it a clear advantage over PNG-24 for transparent images, so that's its other best use case.

WebP is not as good at maintaining sharp edges (such as in text overlays) as PNG-24 and PNG-8, due to the fairly aggressive manner with which it compresses color information in the image. Lossless WebP does not offer quite the file size savings as lossy WebP and therefore is used less often.

Keep in mind, however, that these suggestions are generalized and not prescriptive. It is certainly worth trying out WebP at lower settings, higher settings, and with lossless compression on your images to see how it compares to JPEG and PNG in terms of quality and size.

Browser and tool support

FUN FACT

WebP is a "sister project" to the open source WebM video format.

Lossy and lossless! Binary and alpha transparency! Animation! Smaller files! This little image format really can do it all. Unfortunately, due to an initial lag in browser and tool support, it has taken a while to get established. But it's gaining momentum.

Google first announced WebP in 2010, and it reached its current feature set in 2018. WebP has been supported by all major browsers since 2021 (roughly 97% as of this writing), so that's good news. You can take advantage of the file size savings with confidence, especially if you provide a backup, as we'll discuss soon.

WebP images can be edited and saved in most image editing tools, including Photoshop, Affinity Photo and Designer, GIMP, PaintShop Pro, Pixelmator Pro, Photopea, and many more. Figma cannot export images in WebP format as of this writing, although a plug-in is available at ***oreil.ly/rn4dw***.

See the official WebP site at ***developers.google.com/speed/webp*** for detailed documentation of its compression schemes, updated lists of supporting tools and browsers, and a gallery of samples.

Let's see how WebP handles the croissant photo and ad banner from the previous exercise in EXERCISE 26-2.

EXERCISE 26-2. Take WebP for a spin

As of this writing, to save an image in WebP in Photoshop, you must "save a copy" (it's currently not an option in the Export As panel). The other tools I've tested tend to list WebP along with the other export options.

1. I've opened the original *croissants.jpg* image in Photoshop, selected **File → Save a Copy**, chosen WebP, and saved the image in four variations: lossy WebP with quality levels of 100%, 50%, and 10%, and lossless WebP. Here are the resulting file sizes:

	Original	Lossy WebP (100%)	Lossy WebP (50%)	Lossy WebP (10%)	Lossless WebP
croissants.jpg	299 KB	146 KB	23 KB	18 KB	413 KB

   The WebP versions at 100% and 50% quality are just a little more than half the file size of my JPEGs (277 KB and 42 KB, respectively), so that's an impressive reduction. Looking at the resulting images, I'm pretty happy with the quality at 50%. The 10% quality image is unacceptably blurry. The lossless WebP looks good, but as expected, it comes in at a hefty size just a bit smaller than the lossless PNG-24 (492 KB).

   Conclusion: For photos, a lossy WebP will likely produce a smaller file than a JPEG with similar image quality, making it a good choice. JPEG's only advantage is that it is universally supported by browsers and does not require a fallback.

2. Next, I'm going to save the *ad-banner.png* image as a WebP with the same settings. Here are the resulting file sizes:

	Original	Lossy WebP (100%)	Lossy WebP (50%)	Lossy WebP (10%)	Lossless WebP
ad-banner.png	114 KB	25 KB	7 KB	2 KB	35 KB

   Again, the numbers don't quite tell the whole story, and it's less straightforward than with the JPEG test. The lossy WebPs are much smaller and have better quality (with fewer blotchy artifacts) than the JPEGs at equivalent quality settings (FIGURE 26-15). The lossless WebP is significantly smaller than the lossless PNG-24 in EXERCISE 26-1 as well. However, the PNG-8 I created in the last exercise kept the text the sharpest and had no artifacts in the flat color areas, with a file size smaller than the full-quality lossy WebP (19 KB).

   Conclusion: There are a few decent options here. If I were producing this image for a site, I would play around with WebP quality settings between 50% and 100%. I'd also try reducing the number of colors in the PNG-8 to see if I can get the file smaller while maintaining the quality of the muffin image and the typography. As I mentioned earlier, WebP tends to be a better alternative for photos than for flat graphical images, and that rule of thumb has played out in this exercise.

   I suppose the lesson here is that if you care about making an image as small as it can possibly be, you need to play around with competing formats and settings. Every image is unique, and of course, it will come down to your priorities and opinion.

FIGURE 26-15. Lossy WebP introduces some artifacts around hard edges, but it's not as bad as JPEG at the same quality setting. PNG-8's indexed color model keeps flat areas of color pristine.

Ol' Grandpa GIF

GIF (**Graphic Interchange Format**) was the first image format supported by web browsers, and for a while, it was the *only* file format that would display in a browser window. (I know. I was there.) It was also the only way to add an animated graphic to a web page, making the format synonymous with "animated viral meme." For still images, GIF has lost ground to the PNG-8 and WebP formats, which can do everything GIF can do (and do it better). Even animated GIFs are losing the battle to animated WebPs and MP4 video. Furthermore, newer graphics tools are simply omitting the option to save files in GIF format. Our old friend GIF is heading for retirement. That's okay...we just fight about how to pronounce it anyway.

That said, let's quickly look at what makes GIF tick:

8-bit indexed color

Like PNG-8, GIF is an 8-bit indexed color format. You can save a GIF at lower bit depths, resulting in fewer colors and smaller file sizes.

Lossless compression

GIF compression is lossless, although some image information is lost when the RGB image is converted to indexed color. It uses a compression scheme (called "LZW" for "Lempel-Ziv-Welch") that excels at compressing flat colors.

Binary transparency

GIF images offer only binary (index) transparency, in which pixels are either entirely opaque or transparent.

Interlacing

Interlacing makes a GIF display in a series of passes, similar to progressive JPEGs. Each pass is clearer than the pass before, until the image is fully rendered.

Frame animation

Another feature built into the GIF file format is the ability to display simple animations (FIGURE 26-16). Many of the spinning, blinking, fading, or otherwise moving ad banners you see are animated GIFs. However, animated WebPs and embedded MP4 movie files offer better quality at smaller sizes and are becoming more popular.

FIGURE 26-16. All the frames of this simple animation are contained in one GIF file.

Animated GIFs (and other animated formats) contain a number of animation **frames**, which are separate images that, when viewed in quick succession, give the illusion of motion or change over time, kind of like a flipbook. All of the frames are stored within a single file along with settings that describe how they should be played. The settings include whether and how many times the sequence repeats, how long each frame stays visible (**frame delay**), the manner in which one frame replaces another (**disposal method**), whether the image is transparent, and whether it is interlaced.

There are lots of tools for creating animated GIFs (just do a quick search). Many are web apps that you can use right in your browser or mobile device, and many are free. You can also make an animated GIF in Photoshop by using the Timeline. And if you're ready to move beyond animated GIFs, you'll find several online GIF-to-WebP and GIF-to-MP4 converters.

The JPEG Slayers (AVIF and JPEG-XL)

You're nearly to the end of our web image format roundup; just two more quick introductions to some up-and-coming formats to go. Thanks for hanging in there.

The JPEG format we've known and loved for over 30 years has served us well, but as you've seen, it suffers from blotchy quality loss when compressed aggressively. There have been a few attempts to replace it with formats that offer more efficient compression, with the latest contenders being AVIF and JPEG-XL. Because there aren't many tools that can create them and browser support is recent (AVIF) or extremely limited (JPEG-XL) at the time of writing, I'll do my best to keep this brief. It's good to know that they're out there and that you can use them today as long as you provide JPEG and/or WebP fallbacks.

AVIF

The open source AVIF format was designed to be a replacement for JPEG from the start. Created by the Alliance for Open Media in 2019, AVIF applies the compression scheme from the AV1 video codec to still images (AVIF stands for "AV1 Image File"). The result is file sizes that are on average 50% smaller than JPEG and 20% smaller than WebP, while retaining image quality.

AVIF is generally better than JPEG and WebP at preserving sharp edges and fine details even at high compression rates, so it's useful for photos with a lot of fine detail, sharp edges, and typography (FIGURE 26-17). It also does better at preserving areas of flat color than JPEG.

FIGURE 26-17. AVIF can be used for any image type, but it is a particularly good option when you want to preserve fine detail, straight lines, or crisp typography. (Photos by Jennifer Robbins.)

AVIF is an all-in-one image file format, including these features:

Lossless or lossy compression

Like WebP, you can save AVIF files with lossy (most appropriate for the web) or lossless compression.

Alpha transparency

AVIF files can contain an alpha channel for multilevel transparency. Combined with lossy compression, this will result in a smaller file than PNG-24. Depending on the image type, it may be smaller than WebP as well.

Animation

AVIFs can hold animated frames and are more efficient than animated GIFs.

Color metadata

AVIF can represent wide color gamuts and store HDR images.

AVIF finally achieved support by all major browsers in 2024, so its journey is really just beginning. For now, it is best to provide fallback format options with the `picture` element.

There are a few image editors that can save images as AVIF files as of this writing, such as GIMP and Pixelmator Pro. Adobe tools can open AVIF files but don't currently provide the option to save to that format. Your best bet for now is to use an online image conversion tool that can convert high-quality JPEG images to AVIF, such as Convertio (***convertio.co***). Third-party image optimization services offer AVIF conversion as part of their data-saving strategies as well; we'll talk more about these services in **Chapter 27**.

FURTHER READING

- The Alliance for Open Media has a friendly introduction to AVIF on its blog: "AVIF: Meet the Next Level Image File Format" (***oreil.ly/OyPhl***).
- For an in-depth overview of AVIF features and comparisons of AVIF and JPEG images, see "AVIF for Next-Generation Image Coding" on the *Netflix Technology Blog* (***oreil.ly/k19Hw***).

JPEG-XL

There's a new file format in town, and it's here to beat up all the other formats! Unfortunately, as of this writing, lack of browser and tool support is preventing JPEG-XL from becoming a contender. That may change by the time you read this, so it's worth taking a look at JPEG-XL due to its formidable list of features.

Based on work by Google and Cloudinary, JPEG-XL was released in 2022. Although designed primarily with web use in mind, according to Cloudinary's Jon Sneyers, JPEG-XL is "suitable for the lifecycle of a digital image, from capture and authoring to interchange, archival, and delivery."

As a web image format, JPEG-XL offers the following features, most of which should sound familiar:

- Lossy or lossless compression
- Progressive display
- Wide color gamut and HDR support
- Alpha transparency
- Animation
- Recompression of existing JPEGs without any additional image loss, resulting in 20% smaller files on average (this is a feature not offered by any other image format; see **Note**)
- Better handling of sharp edges, such as text, and compressing areas of flat color

JPEG-XL also has features that make it suitable for high-quality image editing and digital printing. Here are the highlights:

- It can be used as a capture format, like RAW.
- It can contain named layers, selection masks, and multiple channels (like a Photoshop file).
- It can be used with CMYK and spot colors for printing.
- It can capture and store images that are over a *billion* pixels per side (mind-boggling!).

Despite enthusiasm from the development community, as of this writing, only Safari supports JPEG-XL. Without browser support, JPEG-XL may find greater adoption in the image archive and printing arena. Or it may simply fade away.

Similarly, there aren't many mainstream image editors other than GIMP and Affinity Photo/Designer that can save images in JPEG-XL format as of this writing. There are conversion tools out there, including free web-based drag-and-drop converters and bulk converters.

NOTE

*In April 2024, Chrome introduced a more efficient JPEG encoding library (`jpegli`) as a response to JPEG-XL's ability to recompress existing JPEGs without image loss. Read more about it in Google's announcement: **oreil.ly/iekl7**.*

FURTHER READING

- The JPEG-XL Community website (***jpegxl.info***) is the definitive resource for JPEG-XL information. I found the "Battle of the Codecs" page (***jpegxl.info/resources/battle-of-codecs.html***) particularly useful for comparing its features to other formats.
- "The Case for JPEG-XL" by Jon Sneyers (***cloudinary.com/blog/the-case-for-jpeg-x***) provides a compelling overview of JPEG-XL's many unique features.

So, can you use JPEG-XL? Sure! If you have an image that is smaller and looks better when it is saved as a JPEG-XL compared to rival formats, go for it. Just be sure to provide fallback images for browsers that don't support the format.

Summing It Up: Web Image Formats

That completes our tour of JPEG, PNG, WebP, GIF, AVIF, and JPEG-XL. There are a lot of details to keep track of, and I've summarized them in TABLE 26-2 so you can compare features at a glance.

TABLE 26-2. Web image format features

Feature	JPEG	PNG-24	PNG-8	WebP	GIF	AVIF	JPEG-XL
Compression	Lossy	Lossless	Lossless	Lossy or lLossless	Lossless	Lossy or lossless	Lossy or lossless
Colors	16 million+	16 million+	256 (indexed)	16 million+	256 (indexed)	16 million+	16 million+
Transparency	—	Alpha	Binary, Alpha	Alpha	Binary	Alpha	Alpha
Animation	—	APNG format	—	Yes	Yes	Yes	Yes
Browser support	All	All	All	All since 2020 (except IE)	All	All since 2024 (except IE)	Poor support*

* As of this writing, only Safari supports JPEG-XL; support by Firefox and Opera can be enabled with a flag, and Chrome and Edge do not support JPEG-XL. However, that may change by the time you're reading this book, so check *CanIUse.com* for updated support information.

Choosing the Best Format

Now you have a good understanding of what each web-appropriate image format can do, and you're ready to pick the format that will result in the best quality at the smallest file size (see **Note**). TABLE 26-3 provides a more thorough summary of which formats to use for various image types than the teaser I provided at the beginning of this chapter. You'll notice there are quite a few references to providing a fallback image, and in the next section, you'll get a refresher on how to do that.

NOTE

I want to reiterate that for illustrations with flat colors, such as logos or icons, the vector SVG format is the way to go. SVG can also be used for images with a combination of flat colors and a small areas of bitmap image, gradients, or effects like drop shadows. We'll explore it further in **Chapter 28**.

TABLE 26-3. Choosing the best bitmap (raster) file format

If your image...	use...	because...
Is graphical, with flat colors	PNG-8 GIF	The indexed color formats excel at compressing areas of flat colors and can also include fully transparent areas. A PNG-8 is the better choice because it will usually have a smaller file size than a GIF of the same image.
Is a combination of flat colors and photographic imagery	WebP PNG-8 AVIF JPEG-XL	WebP can handle both photographic content and flat color fills. Whether lossy or lossless WebP compression results in the smallest file size depends on the content of the individual image. If you don't mind pixelation (dithering) in the photographic areas, the indexed PNG-8 format may result in the smallest file size. It also provides a good fallback to WebP for non-supporting browsers.
Is a photograph	JPEG WebP (lossy) AVIF JPEG-XL	All of these formats are appropriate for photographic imagery. JPEG is the best-supported, but the others generally result in smaller files with better quality.
Requires multiple levels of transparency	WebP PNG-24 AVIF JPEG-XL	Both WebP and PNG-24 offer alpha channel transparency, with WebP generally resulting in smaller files. You may want to provide a fallback to WebP if you need to support older browsers. AVIF and JPEG-XL also support alpha transparency, but they may be more cumbersome to create and deliver.
Requires animation	WebP GIF AVIF JPEG-XL	WebP and GIF are the safest choices for simple frame animations until browser support improves for the newer formats.
Is the "hero" image for the page, where stunning image quality is critical, or includes sharp edges, like typography	AVIF JPEG-XL (with WebP and JPEG fallbacks)	For an image that represents your brand's first impression, saving it in AVIF and/or JPEG-XL is likely worth the extra effort. These formats retain the best image quality after compression and can represent colors in wide-gamut and HDR color spaces. Provide a JPEG and/or WebP fallback for non-supporting browsers.

A Refresher on Providing Fallbacks with picture

I've mentioned fallbacks frequently in this section, so it's worth a brief refresher on how to provide alternative image formats with the **picture** element. (You first met **picture** back in **Chapter 7**.)

Let's say I have a photograph of a gorgeous New Zealand mountain range that I want to use as the hero image in the header of my home page. I've saved it in JPEG-XL (the smallest) and AVIF (the next smallest) formats to take advantage of the preserved image quality, but I know that a lot of browsers out there won't be able to display them. I can use the **picture** element to make them available to supporting browsers while providing WebP and JPEG fallbacks as well:

```
<picture>
  <source type="image/jxl" srcset="images/mountains.jxl">
  <source type="image/avif" srcset="images/mountains.avif">
  <source type="image/webp" srcset="images/mountains.webp">
  <img src="images/mountains.jpg" alt="">
</picture>
```

The **source** elements provide the location and file type of each image file. The **img** element is required in order for an image to display at all, and it must be the last element in the list. Browsers go through the list and use the first file format that they support (ignoring the rest), so it makes sense to put the **source** elements in order from smallest to largest file size. For this example, a current version of Safari will grab the JPEG-XL version of the image, while Internet Explorer will get to this **picture** element and say, "Nope. Nope. Nope. OK, JPEG! Thank you."

This concludes our exploration of web image formats. They say a picture is worth 1,000 words, but apparently this introduction to image formats required 8,871!

RESOLUTION AND REFERENCE PIXELS

If you are working in a digital medium like the web, you'll need to have a solid understanding of how images and other page elements display on screens, so let's zoom in.

Image Resolution

Image editing programs keep track of how many pixels a bitmap image has per inch. This **pixels per inch** (**ppi**) measurement is the **resolution** of the digital image. On the web, however, the notion of "inches" is irrelevant because the final display size of the image is dependent on the resolution of the screen on which it is displayed (FIGURE 26-18).

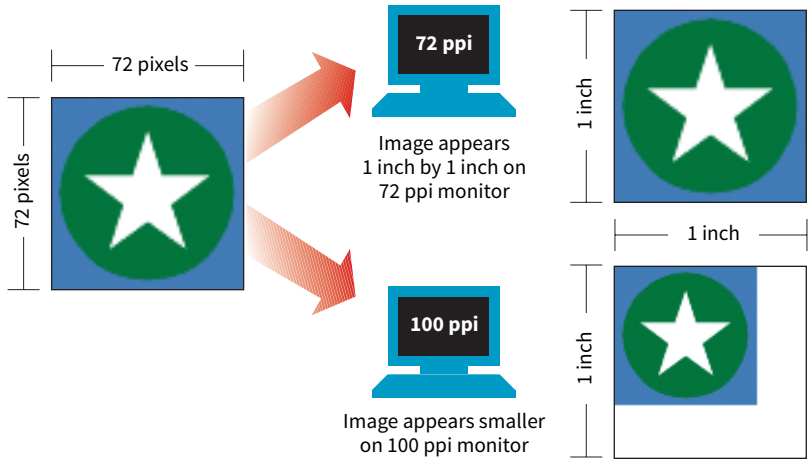


FIGURE 26-18. Inches, and therefore "pixels per inch," are not relevant for digital media, where the size of an image is dependent on the resolution of the screen.

If we're throwing out "inches," we have to toss out "pixels per inch" as well. The only thing we know for sure is that the graphic in FIGURE 26-18 is 72 pixels across, and it will be twice as wide as a graphic that is 36 pixels across. Here's the bottom line: *web images are measured in number of pixels, and the ppi at which they are created is irrelevant*.

That said, web designers tend to work at the default 72 ppi during the design process in bitmap image editors like Photoshop. Working at the default resolution keeps images at roughly the size they'll appear on a standard desktop monitor. You are welcome to create your images at a different ppi, but be sure to be consistent so images don't get resized when you're copying and pasting from one file to another. Note that design tools like Figma, Sketch, and Illustrator are vector-based, so they do not have an inherent resolution, but you can embed bitmap image assets in your design for export later.

Web images are measured in number of pixels. The resolution (ppi) at which they are created is not important.

Screen Resolution

Screen displays are also made up of pixels, so you can measure their resolution in ppi as well. This is referred to as the **pixel density** of the screen (see **Note**).

NOTE

It has become common to use the term "resolution" to refer to the x × y dimensions of a screen. For example, you may see ads for monitors "a resolution of 2,560×1,440." However, for the purposes of our web image discussion, resolution is a measure of pixel density.

The first Macintosh computers had 72 ppi screens, which is pretty crude by today's standards. Early PCs had 96 ppi. These days, most desktop and laptop monitors have resolutions in the range of 109 to 163 ppi. Over the years, manufacturers have been pushing resolution of displays higher and higher, which leads us to...

DPI Versus PPI

The resolution of digital images is measured in pixels per inch (ppi), but printers and printed pages use a different measurement: **dots per inch** (**dpi**), which describes the number of printed dots in each inch of the image. The more ink dots per inch, the sharper the image. The dpi of the printed image may or may not be the same as the ppi of the digital image.

To make things a little more confusing, Android uses the term "dpi" in its technical documentation to refer to device resolutions, when what is actually being described should be identified as ppi. It's common to hear "dpi" and "ppi" used interchangeably, but now you know there is a difference.

High-density displays

From the 1980s to 2010, we could pretty much count on the pixels in our images mapping one-to-one with the hardware pixels in a desktop monitor, as shown in FIGURE 26-18. Of course, there were exceptions—browsers could zoom images larger or smaller on command, and images were scaled down to fit on smartphone screens—but that was the general rule.

There was a seismic shift in 2010, however, when Apple introduced the iPhone 4 with its Retina display. The Retina display packed literally twice the number of hardware pixels into the same physical screen space, giving it a **device pixel ratio** (**DPR**) of 2 (usually indicated as "2x"), resulting in (potentially) much sharper interface elements and images.

The flip side of that, of course, was that the bitmap images we were already using got rendered by twice as many pixels, and could end up looking a bit fuzzy (FIGURE 26-19).

PNG width: 350 pixels
img {width: 350px;}
Standard web images look fuzzy on 2x displays. This PNG is 350 pixels wide in an img element set to 350px wide.
PNG width: 700 pixels
img {width: 350px;}
Images look sharp on 2x displays when they are created at twice the final layout size. This PNG is 700 pixels wide in an img element set to 350px wide.

FIGURE 26-19. Typical web graphics look slightly pixelated on a 2x display

The Retina screen was just the beginning. Apple devices now have 2x or 3x DPRs, and Android devices have 1.5x, 2x, 3x, and even 4x displays. As a result, an actual device pixel may be so small that images and text would be illegibly tiny if they were mapped one to one. What to do?!

Reference Pixels to the Rescue

Thankfully, we have a solution. High-resolution devices use a measurement called a **reference pixel** (or **logical pixel**) for the purposes of layout. Reference pixels enable us to specify sizes without getting caught up in physical pixel densities. Back in **Chapter 19, Responsive Web Design**, we talked about view-

port size. A device's viewport is measured in reference pixels, independent of how many physical device pixels make up the screen.

One reference pixel on a standard 1x screen equals one device pixel. On a 2x screen, a reference pixel is made up of 2 × 2 (4) device pixels; on 3x screens, a point is 3 × 3 (9) device pixels; and so on (FIGURE 26-20).

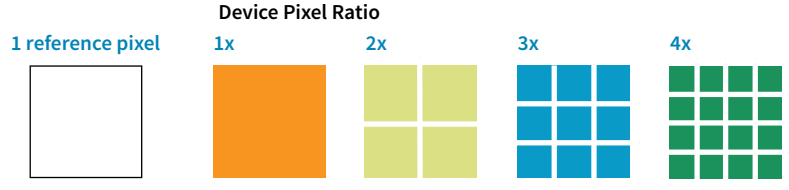


FIGURE 26-20. A reference pixel compared to standard device pixel ratios

Apple calls its reference pixels **points** (**pt**). Android uses the term **device-independent pixels** (**dp**, or the older acronym **dip**). One dp equals one device pixel at 160 ppi (baseline 1x resolution). Android also defines another logical pixel unit called the **scale-independent pixel** (**sp**), but that is used exclusively for specifying text sizes in a user interface, so it is not relevant to our current image discussion. You are more likely to come across the terms pt, dp, and sp when designing UI elements to be used in native smartphone apps.

Pixel Density and the Design Process

What do varying pixel densities mean for the web design and image production process? In your style sheets, you can specify the dimensions of images and other elements in CSS pixels (**px** is technically a type of reference pixel), knowing they will map to the viewport, not the device pixels.

It's when producing the image files themselves that you should consider whether to support higher screen densities. Because 2x displays have become the norm for smartphones and even some laptop and desktop displays, it is common practice to produce images at both 1x and 2x scales and deliver them with responsive image markup (or automate the process). The markup ensures the browser chooses the appropriately sized image and doesn't download more image data than needed.

Using the Frog Prince image in FIGURE 26-19 as an example, to support displays up to 3x, you could create images at 350, 700, and 1,050 pixels wide, knowing that all of them will occupy 350 reference pixels in the layout. In general practice, however, because the human eye can't detect a significant difference in clarity, there's not much benefit to providing versions larger than 2x. Producing responsive image sets is covered in more detail in **Chapter 27**.

Although designers produce assets that cater to high-density displays, most choose to do their design work at 1x scale. The advantages to doing so are listed in the **"Working at 1x Scale"** sidebar.

■ FURTHER READING

For more in-depth explanations of image and screen resolution, I recommend these articles. Ignore the outdated pixel dimensions of specific devices, as those have changed, but the broader concepts still apply:

- "Pixel Density, Demystified" by Peter Nowell (*oreil.ly/irOfE*)
- "Designer's Guide to DPI" by Sebastien Gabriel (*oreil.ly/zmABc*)

It's when producing the images themselves that you need to consider the device pixel ratios you want to support.

Working at 1x Scale

Even if you are creating high-definition versions of your images, it is still recommended that you do your design work at 1x scale. In other words, the pixel dimensions in your working document should match the layout pixels of your design. In Photoshop and other bitmap image editing tools, 1x scale is equivalent to 72 ppi.

The advantages to working at 1x scale include the following:

- It's easier to specify font size and other length measurements as they appear in your working documents without the need to divide everything by two. If you work at 2x and you want 16 pt type, you need to make it 32 pt in your image document. If you want 10 pixels of padding in the layout, you need to create it at 20 pixels, and so on.
- Pixel-snap features work more reliably at 1x. Snapping to even pixels is a way to keep edges crisp in detailed elements such as icons.
- File sizes are much smaller for a design created at 1x, so it is better for performance on your computer. Complex files with lots of artboards and layers at 2x can get sluggish and slow down your work.
- It creates a more realistic sense of how much space you have to work with. A 2x design space might give the impression that you have more room to fit in elements, but they will end up too small and cramped when reduced 50% for 1x displays.
- You can embed high-resolution images in your layout design documents if you need to export larger versions for high-density displays. I'll talk more about this in **Chapter 27**.

Now that we've covered choosing the file format and resolution for web images, there is one more image-related basic you need to have under your belt: transparency. In the next section, you'll become familiar with how binary and alpha transparency work and try your hand at making and saving a transparent image.

WORKING WITH TRANSPARENCY

All web image formats except JPEG allow parts of an image to be transparent so that the page's background color or image shows through. Earlier I mentioned that there are two types of transparency. In **binary transparency** (which you may also see called "index" or "1-bit" transparency), pixels are either entirely transparent or entirely opaque, like an on/off switch. In **alpha transparency** (also called alpha channel transparency), a pixel may be totally transparent, totally opaque, or up to 254 levels of opacity in between, for a total of 256 opacity levels. The advantage of alpha transparency is that the image blends seamlessly with any background color or pattern, as you saw in FIGURE 26-7.

How Binary Transparency Works

Remember that the pixel colors for PNG-8s and GIFs are stored in an indexed color table. Transparency is simply treated as a separate color, occupying one position in the color table. FIGURE 26-21 shows the color table in Photoshop for a simple transparent GIF. The slot in the color table that is set to transparent is indicated by a checkerboard pattern. Pixels that correspond to that position in the color map are completely transparent when the image displays in the browser. Note that only one slot is transparent—all the other pixel colors are opaque.

With binary transparency, there is a risk that the anti-aliased (soft) edges around the image will have a fringe of pixels that don't match the color behind them (FIGURE 26-22). This mismatched fringe is commonly known as a **halo**.

Prevention is the name of the game when it comes to dealing with binary transparency and halos. The trick is to create your image on a background that is close to the background color of the page on which it will appear, so the anti-aliased edges blend to the correct color. Some image editing tools that support web graphic formats provide a way to pick the color to blend with (the **matte color**) when saving or exporting.

Of course, avoiding halos with this method requires that you know the RGB values of the page's background color in advance so you can match the matte color to it. If the page color changes, you need to go back and export the

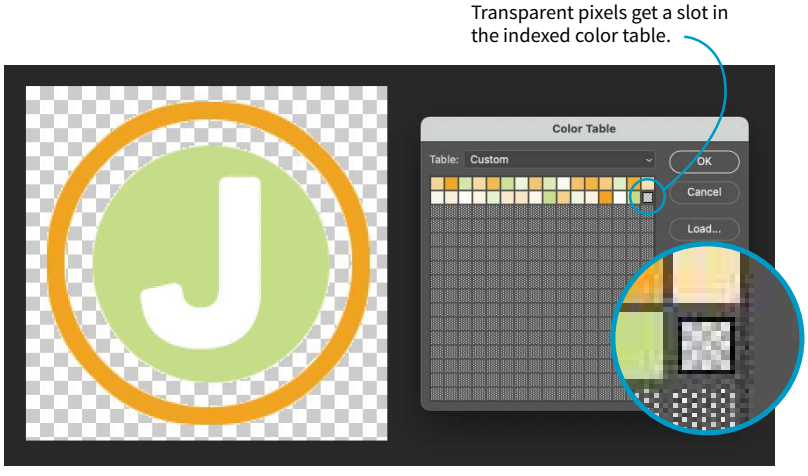


FIGURE 26-21. Transparency is treated as a color in the indexed color table.

graphic again with the new color. That's where alpha transparency has a real advantage—you can change the background, and everything will still blend in perfectly.

How Alpha Transparency Works

RGB images store color in separate channels: one for red, one for green, and one for blue. PNG-24, WebP, AVIF, and JPEG-XL files have the capacity to store a fourth channel, called the **alpha channel**, to store transparency information. In that channel, each pixel may display one of 256 values, which correspond to 256 levels of transparency when the image is displayed. The black areas of the alpha channel mask are transparent, the white areas are opaque, and the grays are on a scale in between. I think of it as a blanket laid over the image that tells each pixel below it how transparent it is (FIGURE 26-23).

PNG-8 "Alpha" Transparency

Technically, indexed PNG-8 files can contain multiple levels of transparency too. Instead of an alpha channel, different levels of transparency are assigned to slots in the color table. Unfortunately, it's difficult to find an image tool that can save a PNG-8 with variable transparency, so although the file format supports it, it's not a practical option. (If you'd like to give it a try, there's an old compression library created by Kornel Lesiński called pngquant, available at *pngquant.org*, that converts PNG-24s to PNG-8s with variable transparency.)

Thankfully, WebP files with alpha transparency rival the small file sizes of transparent PNG-8, so if you need alpha transparency at smaller file sizes than PNG-24, WebP is the way to go.

TERMINOLOGY

Anti-Aliasing

Anti-aliasing is a slight blur applied to rounded edges of bitmap images to make smoother transitions between colors. Aliased edges, by contrast, have stair-stepped edges. Anti-aliasing text and shapes can give your images a more professional appearance.

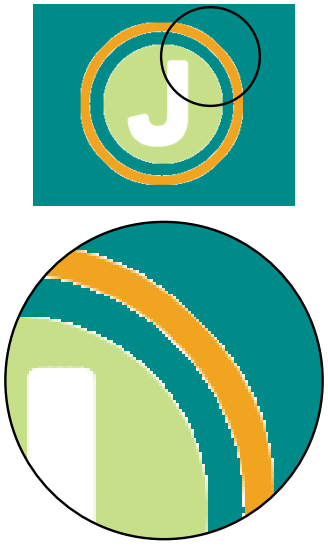
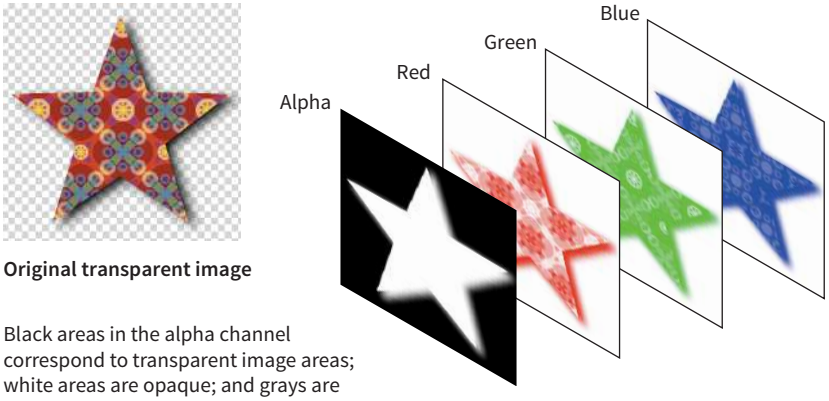


FIGURE 26-22. This GIF with binary transparency has a halo because the semitransparent edges of the original image were blended with a light color that doesn't match the teal background of the page it's displayed on.



Original transparent image

Black areas in the alpha channel correspond to transparent image areas; white areas are opaque; and grays are variable levels of transparency in between.

FIGURE 26-23. Transparency information is stored as a separate channel (alpha).

> **TIP**
>
> Here is a tip for making parts of an existing image transparent while avoiding jagged edges. When selecting the parts of the image that you want to preserve, set 1 pixel of feathering in your selection tool so the selection has a soft edge. Copy and paste the selection into a new transparent document and export it with alpha transparency to preserve the soft edges and allow them to blend with the page background. This tip won't work for every image or page background combination, but it's something to try.

Making Transparent Images

The easiest way to make parts of an image transparent is to design them that way from the start and preserve the transparent areas when you export. Although it is possible to make areas transparent in an existing flattened opaque image, it's usually difficult to get a seamless blend with the background while avoiding halos or jagged edges (see **Tip**).

Instead of just telling you, I'll let you create a layered image and preserve the transparent areas in EXERCISE 26-3. When you create your layered file, be sure that the background layer appears as a gray checkerboard pattern (or similar indication of transparency) and is not filled with a color.

EXERCISE 26-3. Creating a transparent image

In this exercise, we're going to start from scratch, so you'll get the experience of creating a layered image with transparent areas. I'm going to keep it simple, but you can apply these techniques to fancier designs, of course. Again, I'll be using Adobe Photoshop, but the principles apply to any image tool that uses layers (although the specific steps and available export options may differ).

1. Create a new file and make it 250 × 250 pixels with a resolution of 72 ppi. In the New Document dialog box, look for Background Contents and select Transparent from the pop-up menu. Click Create. You should see a square filled with a gray checkerboard pattern indicating the background is transparent.
2. Select the Elliptical Marquee tool and set the Feather setting to 5. Draw a circle in the center of the document and fill it with a color. You should have a circle with blurry edges where the checkerboard shows through. That's all we need for the purposes of this exercise, but you can feel free to add more elements. For fun, I added a letter "J" for Jennifer to mine.
3. Let's first export the image as a PNG-24 with alpha transparency. Select **File → Export → Export As**, select the PNG format under File Settings, and be sure the "Transparency" box is checked (FIGURE 26-24). Also make sure "Convert to sRGB" is checked (you'll need to scroll down in the File Settings to find the Color Space section). Click Export, name the file *circle24.png*, and click Save.

   Note: Another option for this step would be to save the image in WebP format, which also preserves the alpha transparency.

EXERCISE 26-3. Continued

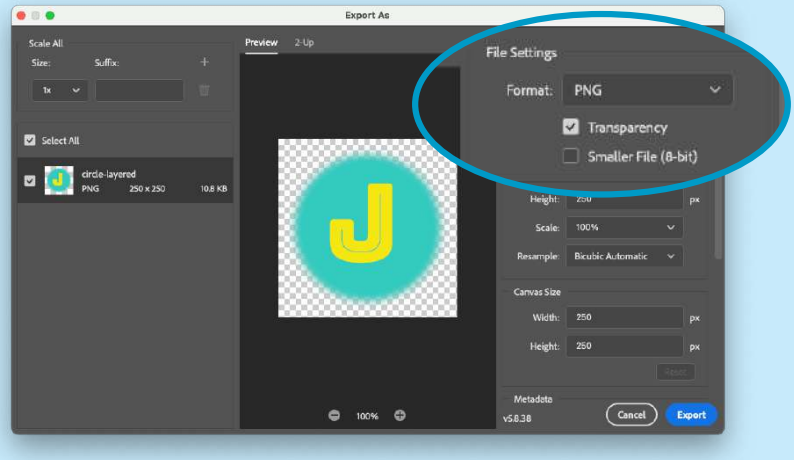


FIGURE 26-24. Exporting a PNG-24 with alpha transparency in Photoshop (2024)

4. Now let's try it again with the indexed image formats that have only binary transparency (see **Note**). Export the image again, but this time select PNG and check "Smaller File (8-bit)." The preview shows that there's now a hard edge around the circle. Save the file as *circle8.png*. Next, repeat the process with the GIF format for comparison. The preview shows that the translucent pixels are blended with white when exported in this format (save this file as *circle.gif*).

How do they look?

Now that I have some transparent images, I'm going to try them out on a minimal web page with a white background. The images and the *transparency.html* file are provided with the materials for this chapter if you'd like to work along. You could also plug in the images you just created.

```
<!DOCTYPE html>
<html>
<head>
  <title>Transparency test</title>
  <style>
    body {background-color: white; text-align: center;}
    img {margin: 2em;}
  </style>
</head>
<body>
  <p>
    <img src="circle24.png" alt="">
    <img src="circle8.png" alt="">
    <img src="circle.gif" alt="">
  </p>
</body>
</html>
```

When I open the file in a browser, the PNG-24 and GIF look more or less the same against the white background (FIGURE 26-25, top). But if I change the background color of the web page to teal (`background-color: #006666;`), the difference between the alpha and

NOTE

For some reason, Photoshop's Export As function treats binary transparency differently for PNG-8 and GIF. PNG-8 uses a 50% threshold to convert transparency levels to either entirely opaque or entirely transparent (resulting in the stair-stepped edge). For GIF, the translucent pixels are blended with a white matte color. I'm including both here because it's a good example of how different tools (or even the same tool!) handle exports differently.

EXERCISE 26–3. Continued

binary transparency becomes very obvious (bottom). The PNG-24 version blends smoothly to either background color thanks to alpha transparency, while the GIF has a pronounced halo. Because the PNG-8 has hard, stair-stepped edges, there's no halo, but it also doesn't have the same soft effect.

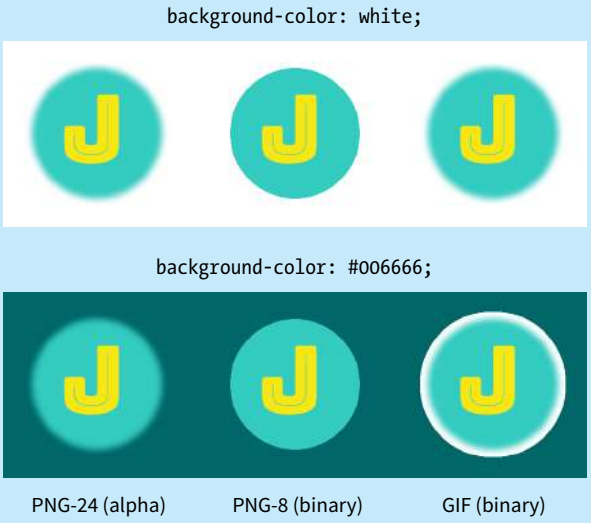


FIGURE 26-25. The difference between binary and alpha transparency becomes very clear when the background color of the page changes. You can also see the difference between the 50% threshold method (PNG-8) and a white matte color (GIF) for handling binary transparency.

Improving binary transparency with a matte color

If you know the background color of your page, you can improve the quality of the transparent indexed images while taking advantage of the smaller file size 8-bit images offer. The trick is filling the translucent pixels with a "matte" color that matches the background color of the page. Many image editing tools offer a matte fill option (see the **"A Matte Alternative"** sidebar if yours doesn't). Curiously, it's not an option when you use the Export As option in Photoshop, but you can do it with the "Save for Web" option.

Although Export As is Adobe's preferred method for exporting images for the web, the legacy "Save for Web" tool offers more options for manipulating and optimizing images. It's listed as "Legacy" because Adobe froze the code for the feature in 2015 and focused on evolving the Export As feature instead. This option is still available, though (at least for now), so let's take a quick look at how it can be used to improve the PNG-8 and GIF.

5. Using the same layered circle document we've been working with, select **File → Export → Save for Web (Legacy)**. In the "Save for Web" panel, you can see a comparison of the source image and a preview with the current settings (you can compare up to four versions). Select PNG-8 from the format options, and the panel displays the settings relative to that format.
6. There are a lot of settings you can adjust here prior to saving the image, but the one that is relevant to this discussion is the matte color. I know that the background color of my page will be **#006666** (teal), so I set my matte color to that value. The preview shows

NOTE

You might be thinking that it would be easier to use a nontransparent image with the same background color as the page background. That may be true, but it can be tricky to get a color in a foreground image to exactly match the same RGB value in the page background on all browsers and platforms. The problem usually lies with the image not using the sRGB color space. Making the edges of the image transparent avoids potential problems.

EXERCISE 26-3. Continued

the edges now blend to that background color (FIGURE 26-26). When placed on the page, this will have the same effect as an image with alpha transparency at a file size smaller than a PNG-24.

Of course, if the background of the page changes, the image will have to be remade.

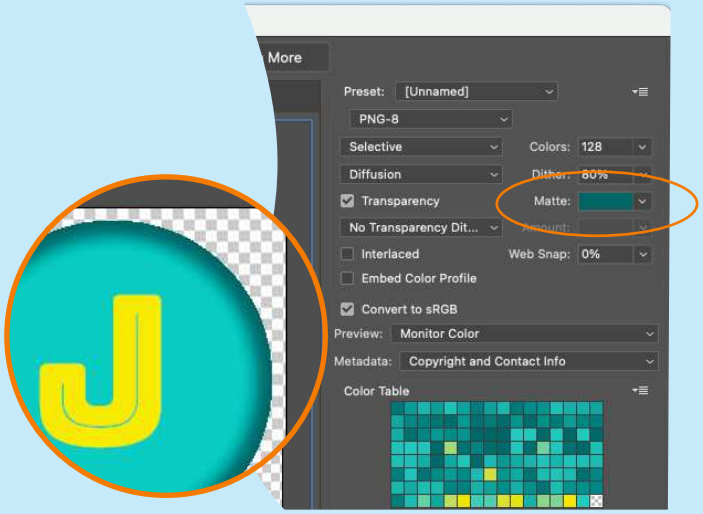


FIGURE 26-26. Using Photoshop's "Save for Web" tool, you can choose a custom matte color to be blended into semitransparent pixels so they match the page background color without any halos.

So many options!

We have a lot of image options after all the exports in this exercise, with several looking more or less the same in the browser. The final choice will come down to file size and quality. Among the images I created, the PNG-8 with the matte fill and the lossy WebP I saved earlier both came in at 9 KB. WebP is the winner here because of the advantages of alpha transparency, but the PNG-8 would make a good fallback image if one is needed. The file sizes of your images will vary, and the results may be different for more complicated images. It's a good idea to check the file sizes and image quality in the Preview to make the best choice.

A Matte Alternative

If your image tool doesn't have the matte feature, create a new layer at the bottom of the layer "stack" and fill it with the background color of your page. When the image is flattened as a result of being changed to indexed color, the anti-aliased edges will blend with the proper background color. Just select that background color to be transparent when exporting to GIF or PNG, and your image should be halo-free.

If selecting a transparent color is not an option in the tool you use, you can copy and paste the important parts of the image, including the blended edges, into a new transparent image file and then export it as a GIF or PNG-8. That's a lot of work for each image, which is why using alpha transparency is a superior choice.

FAVICONS AND SITE ICONS

As long as we're talking about images, there is one last website-related image to cover: the favicon. A **favicon** is the little icon that shows up to the left of the page title in the browser tab and in bookmark lists (FIGURE 26-27). First introduced as an Internet Explorer 5 feature in 1999, favicons were quickly adopted by other browsers. Providing them isn't required, but it is a best practice for helping users identify and find your site in a long lineup of tabs or bookmarks; they're a little attention to detail that strengthens your brand.

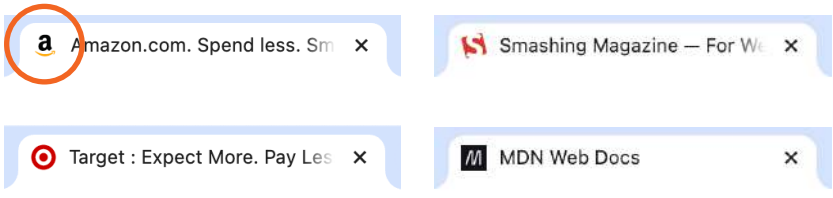


FIGURE 26-27. Favicons for Amazon.com, Target.com, SmashingMagazine.com, and developer.mozilla.org in browser tabs (shown in Chrome on macOS)

Other web-enabled devices use site-associated icons that are similar to favicons. For example, Apple's iPhone and iPad represent sites and web apps with an icon (called a **touch icon**) when you save them as a shortcut on the home screen. Android uses icons for sites installed as system applications (**progressive web apps**, or **PWAs**).

This section introduces what it takes to create a favicon as well as a full icon set that covers all the bases.

Favicon Image Formats

Traditionally, favicon images needed to be saved in the ICO format with the filename *favicon.ico*, and that is still the only format that works for legacy (out-of-date) browsers and Internet Explorer 10 and earlier. Modern browsers also support favicons in SVG and PNG formats (see **Note**).

NOTE

Browsers also support JPEG, but that is not an appropriate file format for the type of image typically used for tiny icons.

The ICO format

ICO files can contain multiple images, which means you can provide the icon at multiple sizes in one file. Icons are generally provided at 32×32 pixels and 16×16 pixels. It is possible to provide just a 32-pixel-square version and let it scale down to 16 px automatically, but if the image quality suffers unacceptably, providing a dedicated and fine-tuned 16-pixel image is a good idea. You may also choose to include a 48-pixel-square version for high-resolution displays.

The open source image editor GIMP can save layered images in ICO format natively. If you use another tool, the easiest thing to do is create your icons and convert them to ICO with a separate tool. There are several free drag-and-drop ICO converters online, such as ***icoconverter.com*** and ***iconconvert.com***.

PNG and SVG

PNG images are easier to create with mainstream design tools and also allow transparent areas in the icon.

SVG offers the advantage of being scalable without loss of quality, eliminating the need to create multiple bitmap versions at various sizes. Because it is an XML file that can contain its own style sheet, you can use media queries to alter the icon design on the fly based on browser conditions, such as switching icon colors when the browser is in "dark mode." SVG is not universally supported as a favicon format, however, so use it along with a traditional *favicon.ico* as a fallback.

How you add your favicon to the site depends on the file format and what you are trying to achieve. We'll look at the options for that next.

The Old Favicon-in-the-Root-Directory Trick

If you are only interested in providing a favicon for browsers and aren't concerned with mobile icon sets, the traditional favicon process is easy:

1. Save your icons in ICO format and name the file *favicon.ico*.
2. Put that file in the root directory of the site, where browsers know to look for it.
3. There is no third step. That's it!

All browsers know to look in the root directory for *favicon.ico*, so this method takes care of everyone.

Adding Favicons with the link Element

Favicons in any format can be provided using the **`link`** element in the **`head`** of the document. This is the W3C's preferred method for adding favicons, although it is not supported by older browsers and Internet Explorer (provide a *favicon.ico* file in the root as a fallback). Here is an example of the markup:

```
<link rel="icon" type="image/png" href="/favicon-32x32.png" sizes="32x32">
<link rel="icon" type="image/png" href="/favicon-16x16.png" sizes="16x16">
<link rel="icon" type="image/svg+xml" href="/favicon.xml">
```

The **`rel`** attribute tells the browser that the image is to be used as the favicon. The **`type`** attribute identifies the image file type and is required if the format

is something other than ICO. The **`href`** attribute provides the location of the file. It is best practice to put all icon files in the root directory (alongside *index.html*), but it is also possible to put them in an "images" directory as long as that is reflected in the pathname. Finally, the **`sizes`** attribute defines the size of the icon. It is not required for SVG images as they can be used at multiple sizes.

Providing a Full Icon Set

You may decide to go the extra mile and create a complete icon set to represent your site on desktop browsers, iOS, and Android. In that case, you will need to create a package with a set of images, a stack of **`link`** elements, and a special file called a "web manifest" for Android devices. You have a few options here as well.

Do it yourself

Andrey Sitnik (who is also the creator of the popular PostCSS and Autoprefixer tools) proposes a process for creating the minimum number of files and markup in his article "How to Favicon in 2025: Three Files That Fit Most Needs" (***oreil.ly/vSg_5***). My summary here is based on his approach, but I recommend reading the full article for more valuable details and techniques.

The files that go in the root directory are:

- An ICO file (*favicon.ico*) containing a 32×32 (and optionally a 16×16) version of the icon for legacy browsers
- An SVG for modern browsers
- A 180×180 PNG used for the Apple touch icon (named *apple-touch-icon.png*)
- A 192×192 PNG for the Android web app icon on the home screen
- A 512×512 maskable icon for Android launchers (keep the content of this icon to the 409 × 409 area in the middle of the image, leaving the edges clear so it can be cropped as needed by the launcher)
- A 512×512 PNG for the Android web app splash screen
- A web app manifest text file that tells Android devices how to install your site as a PWA (named *manifest.webmanifest*)

This is the **`link`** stack that goes in the **`head`** of your HTML documents:

```
<link rel="icon" href="/favicon.ico" sizes="32x32">
<link rel="icon" href="/icon.svg" type="image/svg+xml">
<link rel="apple-touch-icon" href="/apple-touch-icon.png">
<link rel="manifest" href="/manifest.webmanifest">
```

The **web manifest** is a **JSON** (JavaScript Object Notation, a format for storing and transporting data) file that includes the pointers to the 192- and 512-pixel-square icons as well as the maskable icon for Android. This is the content of the JSON file:

```
// manifest.webmanifest
{
  "icons": [
    {"src": "/icon-192.png", "type": "image/png", "sizes": "192x192" },
    {"src": "/icon-mask.png", "type": "image/png", "sizes": "512x512",
"purpose": "maskable" },
    {"src": "/icon-512.png", "type": "image/png", "sizes": "512x512" }
  ]
}
```

Favicon set generators

If you aren't down for creating all your icons and code from scratch, you'll be glad to know there are favicon generators that will do it for you. They offer a real convenience, but the downside is that some may not be updated to the latest best practices and may produce more files than necessary. That said, here are a few all-purpose generators that do a nice job:

RealFaviconGenerator (realfavicongenerator.net)

The RealFaviconGenerator lets you preview and tweak your icons for every device and purpose. It's a great visual tool that allows a lot of customization. It also integrates with common web development build tools. An SVG favicon generator is also available (***realfavicongenerator.net/svg-favicon***).

Favicon.io by John Sorrentino (favicon.io)

For a streamlined experience and the minimal number of files to meet the needs of current browsers, try Favicon.io. It is similar to Andrey Sitnik's process outlined in the previous section, but uses two PNG files instead of SVG.

Designing Icons

Designing a clear and identifiable icon is an art. Here are some guidelines to get you started (the **"Further Reading"** sidebar lists a few articles with more detailed advice):

- **Keep it simple.** Build your icons from simple geometric shapes and avoid small details that will blur at small sizes.
- **Start with a vector original.** Export bitmap versions at the required sizes.
- **Size bitmaps incrementally.** If you are starting with a bitmap image, scale them down in increments and check the quality at each step.

■ COOL TOOL

If you want to use just the initial letter of your site's name as a favicon, *Favicon.io* generates icons based on your character, font, and color selections.

- **Fine tune at the pixel level.** For very small icons (32 and especially 16 pixels square), you'll likely need to edit the pixels manually to get the best result.
- **Include margins on Apple touch icons.** Apple recommends that icons include a 10% margin on all sides, with the icon image centered in the remaining 80% of the space. This prevents parts of the image from being hidden when rounded edges or other cropping is applied.
- **Try using one detail.** If your logo is complicated, consider using just a distinctive detail, as O'Reilly Media does (FIGURE 26-28).

FIGURE 26-28. O'Reilly Media uses a distinctive detail from its logo in its favicon.

■ FURTHER READING

The following articles provide useful information about icon design in general that will serve you well for creating favicon sets:

- "Icon Design: A Beginner's Guide [+ Resources]" by Flori Needle on the *HubSpot* blog (*blog.hubspot.com/website/icon-design*)
- "6 Easy Steps to Better Icon Design" by Scott Lewis on *Smashing Magazine* (*oreil.ly/LAs5L*)
- "Here's How to Design an Icon from Scratch: A Step-by-Step Guide" by Maria De La Riva on the *CareerFoundry* blog (*oreil.ly/bEmZB*)

THAT'S A WRAP ON IMAGE BASICS!

We've covered a lot of ground in this chapter. You should now have a good foundation in web images, including where to find an image and what file format to save it in. You've learned about image and screen resolution as well, and you got some hands-on experience with transparent images. You also know how put a favicon in the browser tab. In the next chapter, we'll talk about the image asset production process as a whole, including tips for optimization, responsive images, and automation—but first, a little quiz.

TEST YOURSELF

Answer the following questions to see if you got the big picture on web images. You can check your answers in **Appendix A**.

1. What is the primary advantage of using rights-managed images?

2. What does "ppi" stand for?

3. How would you ensure an image stays sharp on a display with a DPR of 2x?

4. What is "indexed color"? What file formats use it?

5. Name two things you can do with a WebP that you can't do with a JPEG.

6. Name one thing you can do with a WebP that you can't do with a PNG.

7. Name one thing you can do with a PNG that you can't do with a GIF.

8. JPEG's lossy compression is cumulative. What does that mean? Why is it important to know?

9. What is the difference between binary and alpha transparency?

10. What file format(s) could you convert a JPEG to in order to get a smaller file?

11. How do you give browsers multiple formats to choose from using HTML markup?

12. What is an ICO file used for? Where should you put it?

13. Which formats are the best match for the image in FIGURE 26-29?

a. JPEG	b. PNG-24	c. PNG-8	d. GIF
e. WebP	f. JPEG-XL	g. AVIF	h. SVG

FIGURE 26-29. Which image formats would be best for this image? (©2024, Arlo Robbins.)

14. Which formats are most appropriate for the image in FIGURE 26-30?

a. JPEG	b. PNG-24	c. PNG-8	d. GIF
e. WebP	f. JPEG-XL	g. AVIF	h. SVG

FIGURE 26-30. Which image formats make sense for this image? (Photo by Jennifer Robbins.)

CHAPTER
27

IMAGE PRODUCTION TECHNIQUES

IN THIS CHAPTER

Image asset production strategy

Image optimization

Responsive image set approaches

Image production tips and techniques

Image automation services

Throughout this book, you've gotten to know the important principles of performance and responsive web design. Let's put all of this know-how together in a strategy for approaching image production.

IMAGE PRODUCTION STRATEGY

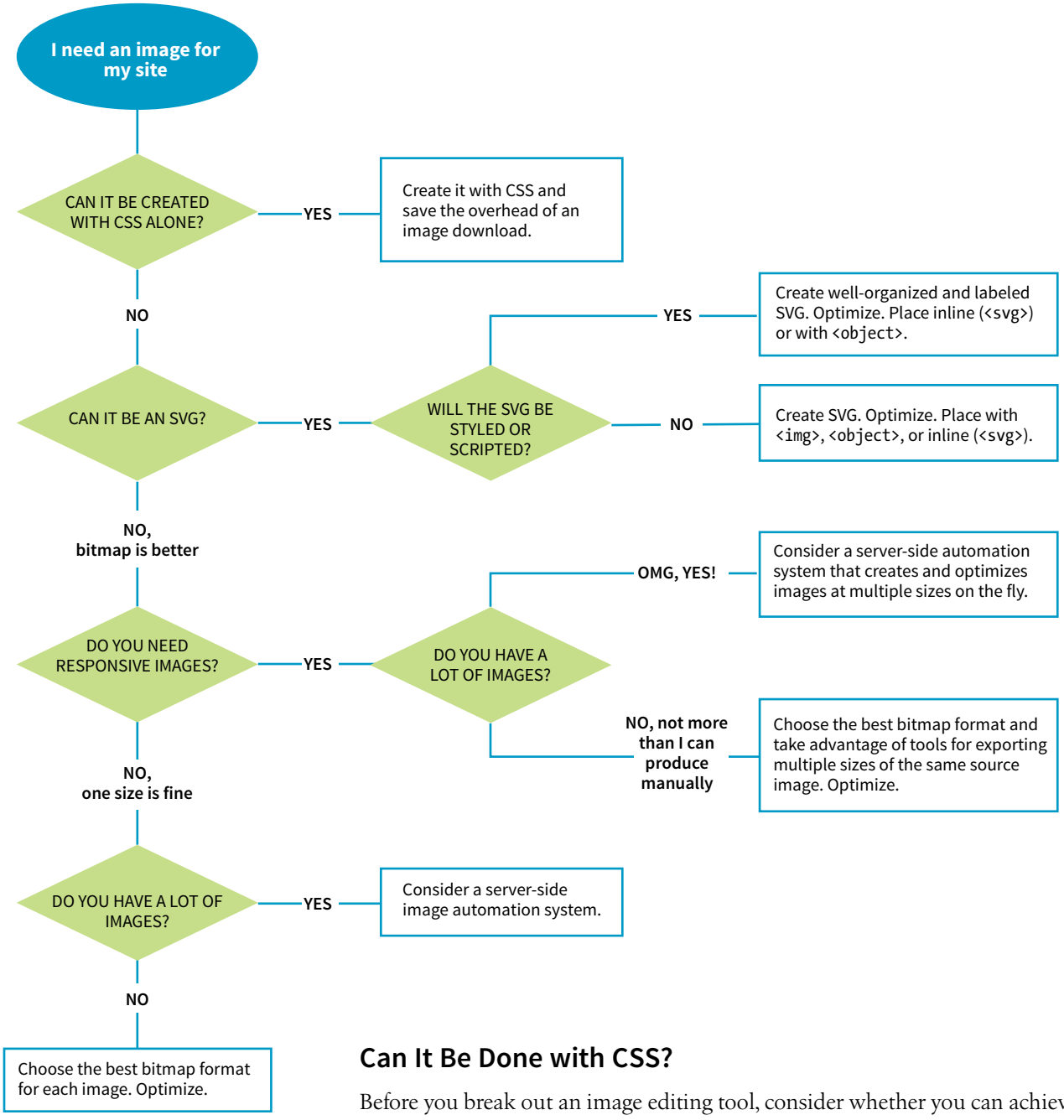
As a conscientious web designer concerned with providing the best experience across a wide range of devices, you should have these priorities in mind when handling images for a site:

- Keeping the file sizes of images as small as possible
- Minimizing the number of HTTP requests to the server
- Not downloading more image data than is needed for devices with smaller screens
- (Optional) Delivering high-quality images to high-density displays

It may be helpful to approach these image requirements systematically, ruling out classes of images and unnecessary tasks. FIGURE 27-1 diagrams a series of questions you can use to cull your image production options. I'll be using this flowchart as the jumping-off point for discussions on optimization, responsive image production, efficient production techniques, and automation.

First off, let's determine whether you need an image at all.

FIGURE 27-1. This flowchart is intended to help narrow down the options for image asset production. It is the foundation of the discussions in this chapter.


I need an image for my site
CAN IT BE CREATED WITH CSS ALONE?
YES
Create it with CSS and save the overhead of an image download.
NO
CAN IT BE AN SVG?
YES
WILL THE SVG BE STYLED OR SCRIPTED?
YES
Create well-organized and labeled SVG. Optimize. Place inline (<svg>) or with <object>.
NO
Create SVG. Optimize. Place with <img>, <object>, or inline (<svg>).
NO, bitmap is better
DO YOU NEED RESPONSIVE IMAGES?
YES
DO YOU HAVE A LOT OF IMAGES?
OMG, YES!
Consider a server-side automation system that creates and optimizes images at multiple sizes on the fly.
NO, not more than I can produce manually
Choose the best bitmap format and take advantage of tools for exporting multiple sizes of the same source image. Optimize.
NO, one size is fine
DO YOU HAVE A LOT OF IMAGES?
YES
Consider a server-side image automation system.
NO
Choose the best bitmap format for each image. Optimize.

Can It Be Done with CSS?

Before you break out an image editing tool, consider whether you can achieve what you're after with CSS alone. Not only will the effect be a fraction of the file size, but you'll also avoid another call to the server.

Effects like rounded corners and gradients that once required images are achievable with CSS properties (**`border-radius`** and **`radial-gradient`**/**`linear-gradient`**, respectively).

It is also possible to make little drawings with CSS, which may be useful in place of bullets and icons. Basic shapes such as circles, rectangles, triangles, and more can be created with empty `div` elements and some trickery with borders and transforms. To read how the shapes in FIGURE 27-2 (left) were created, see "The Shapes of CSS" by Chris Coyier (*css-tricks.com/the-shapes-of-css/*).

Some people have created amazingly complex illustrations using HTML and CSS (FIGURE 27-2), but the technique, which had its heyday around 2010–11, is largely for demonstration purposes and not recommended for production. If you are curious, see "95+ CSS Illustrations," a collection of inspirational CSS drawings, many of which include demos and code so you can see how they are made (*freefrontend.com/css-illustrations/*). You will also find a lot of CSS illustrations on *codepen.io*.

If you need more than a CSS effect, it's time to think about image formats.

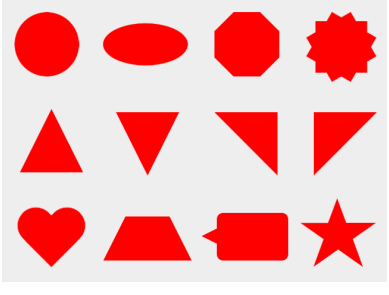
Can It Be an SVG?

If your image is a logo, icon, or other illustration, creating it in a vector drawing tool and saving it as an SVG offers the benefits of resolution independence and a potentially smaller file size. Now that browser support is reliable, SVG is a great option for responsive images.

If you place the SVG code inline with the `svg` element, you save another HTTP request and gain the ability to style and script the element. Or, if a static illustration is all you need, embedding the SVG in the document with the `img` element is a perfectly fine option.

In **Chapter 28, SVG**, you'll take a long journey through the SVG format, so I won't say much more here other than that SVG should be your first choice if you can create the image or illustration in vector format.

If SVG is not appropriate for your image, then you may need to go with a bitmap format. There are still a few things to consider.



Shapes by Chris Coyier
css-tricks.com/the-shapes-of-css

Animated figure by Julia Miocene
codepen.io/miocene/pen/zYmLOJY

Cheesecake by Sasha Tran
codepen.io/sashatran/pen/ggGeZr

FIGURE 27-2. These little drawings are created with HTML markup and CSS alone.

What Is the Best Bitmap Format?

As you learned in **Chapter 26, Web Image Basics**, image format has a large impact on file size, so choosing the most appropriate format for your image is the first step to optimization. You may also choose to save certain images in multiple image formats and deliver them with the `picture` element to take advantage of the image quality and file size savings of newer formats.

Regardless of the format you choose, there are steps you can take to make sure the file is as small as it can be, whether during the design process, on export, or using a special image optimization tool. I cover all of these options in the upcoming section **"Optimizing Images."**

With the format decided, it's time to start thinking about how many versions of each image you need to create.

Does Your Layout Require Responsive Images?

The next thing to consider is whether your layout requires responsive images. To recap, "responsive images" refers to the practice of providing a set of images and letting the browser pick the one best suited to the user's viewing environment. Responsive image techniques address these scenarios, which I'll discuss further in **"Responsive Image Sets"** later in this chapter:

- Preventing browsers on small screens from downloading more image data than they need
- Providing images large enough to look crisp on high-density displays
- Providing alternative versions of the image based on content (art direction)
- Providing alternative file formats (see **Chapter 8, Adding Images**, and **Chapter 26** for details)

No, one size is fine

Some pages, such as text-heavy pages with small illustrations, might get by fine with just one version of each image that serves all screen sizes. In those cases, save or export your image in the most appropriate file format and make sure it's optimized, and you're done.

Yes, I need each image saved at multiple sizes

If you're going the responsive images route, things start getting interesting. The next question is...

Does Your Site Have a Lot of Images?

No, I can handle images manually

If your site has a reasonable number of images that are updated on a reasonable schedule, you should be able to produce them yourself and upload them to the server (as I do for ***learningwebdesign.com***). The good news is that there are a whole slew of new tools designed especially to support the web image asset production process, including ways to create several versions at once and to optimize them in batches. We'll look at some best practices for image production in the **"Tools, Tips, and Techniques"** section later in this chapter.

Yes, my site has a ton of images (hint: automate it!)

Although it's terrific that we have HTML markup for getting the right images to the right browsers, the current system is a bit cumbersome, requiring stacks of code and the need to produce multiple images. If you're working on an image-heavy site, it could prove to be unmanageable. Image processing is a task that begs to be automated. The solution: let a *server* do it! You'll learn more about image resizing and optimization services in the upcoming section **"Automate It!"**

Summing Up Image Strategy

To put it concisely, if you can't achieve the effect you want with CSS or use an SVG, then save a copy (or copies, if you need responsive images) of the image in the best bitmap format(s) for the image type. If are there are more images than you can produce manually, look into image optimization software and services. You should also make sure that all images are optimized to their smallest file size, so let's get right to some techniques for keeping file sizes small beyond simply choosing the best image format.

OPTIMIZING IMAGES

Because a web page is published over a network, it needs to zip through the lines as little packets of data in order to reach the end user. It is fairly intuitive, then, that larger amounts of data require a longer time to arrive. And guess which part of a standard web page packs a whole lotta bytes? That's right, the images. In fact, images make up roughly 60% of the data on the web.

If you study the flowchart in FIGURE 27-1, you will see that all paths end with "Optimize" (see **Note**). Making your image files as small as they can be is critical for fast-loading sites, so all web designers and developers should have multiple image optimization tricks up their sleeves.

NOTE

The paths in the flowchart may end with optimization, but I'm putting this section first because if you only read half of this chapter, this is the information I want you to know!

All web designers should have multiple image optimization tricks up their sleeves.

Optimization approaches fall into three broad categories:

- Efforts you make manually and deliberately during the design and export process
- Use of post-export compression tools that root through the code and crunch the file size down even further, generally by throwing out unused data
- Use of automated image optimization services (these will be discussed in **"Automate It!"**)

This section starts with general guidelines for limiting file size. Next, because each image format is slightly different under the hood, we'll examine optimization strategies that cater specifically to lossy and indexed formats (see **Note**). Finally, I've rounded up some optimization tools that are a good last step in any image production process.

NOTE

*Of course, it is important to optimize SVGs as well, but I've saved that discussion for the SVG chapter (***Chapter 28***).*

General Optimization Tips

Regardless of the image format, there are a few basic strategies to keep in mind for reducing the impact of images on page display and the overall user experience. In the broadest of terms, they are as follows:

Choose the appropriate image format.

But you already knew that.

Size images appropriately.

There's no point asking users to download a giant image if it occupies only a small space in the layout. Take the time to resize images appropriately for your layout. A more sophisticated approach is to create the images at a variety of sizes and use responsive image markup to let the browser pick the best size. I'll introduce strategies for this in **"Responsive Image Sets."**

Limit the dimensions.

Although fairly obvious, the easiest way to keep file sizes down is to limit the dimensions of the images. There aren't any magic numbers; just don't make images any larger than they need to be. By simply eliminating extra space in the graphic in FIGURE 27-3, I was able to reduce the file size by 3 KB (23%).

Run the image through an optimizer.

You should have a number of image optimization tools at your disposal. I'll list several throughout this section, many of which are free to use.

Reuse and recycle.

If you use the same image repeatedly in a site, it is best to create only one image file and point to it repeatedly wherever it is needed. This allows the browser to take advantage of the cached image and avoid additional downloads.

FIGURE 27-3. You can reduce the size of your files by cropping out extra space.

Take advantage of "lazy loading."

In **Chapter 8**, I introduced lazy loading, which is a technique that defers the loading of images until they are visible on the page. Adding `loading="lazy"` to an `img` element that is farther down in the scroll ensures that waiting for images that can't be seen on initial page load doesn't slow down the display of the page.

Catering to Compression Schemes

You got to know the inner workings of the popular image formats in the previous chapter. Now you can use that understanding to make your images smaller. Let's look at approaches for optimizing each web image format.

Optimizing JPEG

Here are the general strategies for reducing the file size of JPEGs:

- Be aggressive with compression.
- Choose "progressive" if available.
- Choose "optimized" if available.
- Blur the image.

Be aggressive with compression.

Your number one tool for optimizing JPEGs is the quality setting that you'll find in every image editing tool. This allows you to adjust the rate of

Search Engine Optimization (SEO)

Another thing to consider when creating your images is how they can help Google and other search engines find your content. Here are some tips:

- Give your image files descriptive names. Including useful keywords in an image's title helps with SEO. For example, *london-rose-garden.jpg* is better than *IMG90063.jpg*.
- Use descriptive "alt" text. In addition to being an accessibility best practice, descriptions provided with the `alt` attribute in the `img` element are the most important tool for getting your images to show up in image search results.

compression: lower quality means higher compression and smaller files. If your image editor has a preview, you can keep an eye on the image quality while changing the compression level. Different images can withstand different amounts of compression, but in general, images hold up reasonably well at moderate (50–70%) and even low (30–40%) quality settings. The exact results at particular settings vary from program to program, so choose a setting that produces the best balance of quality and file size for your particular image.

The best compression level is based on the particular image and its purpose on the site.

Choose "progressive" if available.

Progressive JPEGs display in a series of passes, starting with a blurry image that gets sharper with each pass. Not only do progressive JPEGs generally have smaller file sizes, they also speed up the perceived time that the image takes to display because you're giving the user a hint about the image that's on its way immediately. Large images get the most benefit from being saved in progressive format.

Choose "optimized" if available.

Optimized JPEGs have slightly smaller file sizes and better color fidelity than standard JPEGs (although I've personally never been able to see the difference). For this reason, you should select the "optimized" option if your image software offers it.

Blur the image.

Because soft images compress smaller than sharp ones, you can try applying a slight blur or "smoothing" to the image to give the lossy compression something to chew on. Even an imperceptible blur over the whole image can reduce file size.

You might also choose to apply a more aggressive blur to less important areas of the image while preserving areas of interest. In FIGURE 27-4, I applied a blur to all areas of the image except the face, which remains at the original quality, and reduced the file size by 6 KB, or 23%.

26 KB
Quality: 60%, no blur

20 KB
Quality: 60%, Gaussian blur applied to areas except the face

FIGURE 27-4. Applying blur to less important parts of an image can help reduce the size of the exported JPEG.

"Optimizing" PNG-24

PNG's lossless compression is good at preserving image quality, but the same image will always be smaller when it is saved as a lossy JPEG or WebP. Therefore, your first "lean and mean" strategy for photographs is to avoid PNG-24 and go with JPEG or WebP instead.

You may be using PNG-24 because you need multiple levels of transparency (a valid reason). If that is the case, you have a few options:

- Run the image through one of the image optimizers listed later in this section to strip out useless metadata but preserve the image itself. This option will likely result in only modest file size savings.
- Convert the image to WebP format to preserve the alpha transparency and reduce the file size.

- Convert the image to PNG-8 and use binary transparency, possibly with a matte color to preserve soft edges.

Optimizing PNG-8 and GIF

For the indexed color formats, PNG-8 and GIF, follow these optimization strategies when creating the image and during the export process:

- Reduce the number of colors (bit depth).
- Design with flat colors.
- Avoid or reduce dithering.

Reduce the number of colors (bit depth).

The most effective way to reduce the size of an indexed color image is to reduce the number of colors in the image. The number of colors is also referred to as the image's **bit depth** (see the **"Bit Depth"** sidebar).

Although PNG-8s and GIFs can contain up to 256 colors, there's no rule that says they have to. Reducing the bit depth significantly reduces the file size. One reason is that files with lower bit depths contain less data. Color reduction also creates more areas of flat color by combining similar, abutting pixel colors. More flat color areas mean more efficient compression.

In many tools, you can select the number of colors when converting the image from RGB to indexed color manually. This is usually done in a separate step prior to exporting the file to PNG-8 or GIF. Photoshop's "Save for Web" tool allows you to choose the bit depth in the format options (FIGURE 27-5) and preview the resulting image and color palette just before saving the file.

Bit Depth

Bit depth is a way to refer to the maximum number of colors a graphic can contain. This chart shows the number of colors each bit depth can represent:

1-bit	2 colors
2-bit	4 colors
3-bit	8 colors
4-bit	16 colors
5-bit	32 colors
6-bit	64 colors
7-bit	128 colors
8-bit	256 colors

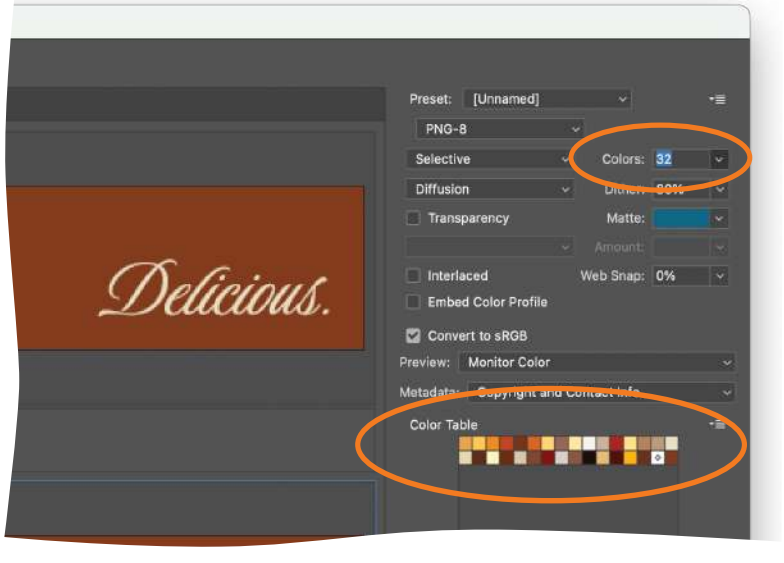


FIGURE 27-5. Reducing the number of colors in Photoshop's "Save for Web" tool

If you reduce the number of colors too far, of course, the image may begin to fall apart, or cease to communicate effectively. For example, in FIGURE 27-6, once I reduced the number of colors in the PNG to eight, I lost the rainbow, which was the whole point of the image. This "meltdown" point is different from image to image. (Granted, this barn and sky image should be a JPEG, but it demonstrates the effects of optimization dramatically, so thank you for bearing with me.)

You'd be surprised how many images look perfectly fine with only 32 pixel colors.

That said, you'll be surprised to find how many images look perfectly fine with only 32 pixel colors (a 5-bit bit depth). That's usually my starting point for color reduction, and I go higher only if necessary. Some image types fare better than others with reduced color palettes, but as a general rule, the fewer the colors, the smaller the file.

Design with flat colors.

Choosing flat colors over gradients and patterns makes a big difference in file size for images saved in indexed formats, as shown in FIGURE 27-7. Reducing the colors from 256 to 8 goes a long way in reducing the file size for the KAPOW! graphic, but the colors in the blend are approximated with a dither pattern that breaks up the flat color areas. The version with flat colors is half the size of the dithered version, even though both images have been reduced to 8 colors.

I feel obliged to say that images like this one should be drawn with vectors and saved in SVG format, which will be smaller and more versatile than bitmap versions. However, if you have a reason for saving PNGs, design them in a way that takes advantage of the compression scheme.

256 colors (**21 KB**)

64 colors (**13 KB**)

8 colors (**6 KB**)

FIGURE 27-6. Reducing the number of colors in an image reduces the file size but can also degrade it significantly..

13.5 KB
PNG with gradient blends and 256 colors

6.8 KB
PNG with dithering and only 8 colors

3.7 KB
PNG designed with flat colors and 8 colors

FIGURE 27-7. For PNG-8 and GIF, you can keep file sizes small by replacing gradients and patterns with flat colors.

Reduce dithering.

When the colors in an RGB image are reduced to a specific palette, the colors that are *not* in that palette get approximated by a speckle pattern called **dithering**. In photographic images, dithering is not a problem and can even be beneficial; however, dithering in flat color areas is usually undesirable. It's also undesirable in terms of optimization, because the speckles disrupt otherwise smooth areas of color and stand in the way of efficient compression.

Photoshop, GIMP, and other image editors allow you to specify whether and how the image dithers when converting to indexed color. Turning dithering off entirely may result in a banding effect, as shown in FIGURE 27-8. If that is unacceptable, turn the dithering back on or try a higher bit depth.

Dithering: **9.6 KB**

No dithering: **7.8 KB**

FIGURE 27-8. Turning off or reducing the amount of dithering reduces the file size but can result in banding. Both images have 32 pixel colors.

Optimizing WebP

Your primary tool for optimizing WebP images is the quality setting (FIGURE 27-9). Some tools give you a scale from 1 to 100; others just use descriptive settings like "medium" and "best." The quality scale for WebP is not the same as for JPEGs—for example, a 50%-quality WebP will not look the same as a 50%-quality JPEG. In general, it's best to stick with higher settings for WebP because image integrity drops off quickly as you go lower. Even at higher quality settings, a WebP image will generally be smaller than a JPEG with comparable image quality.

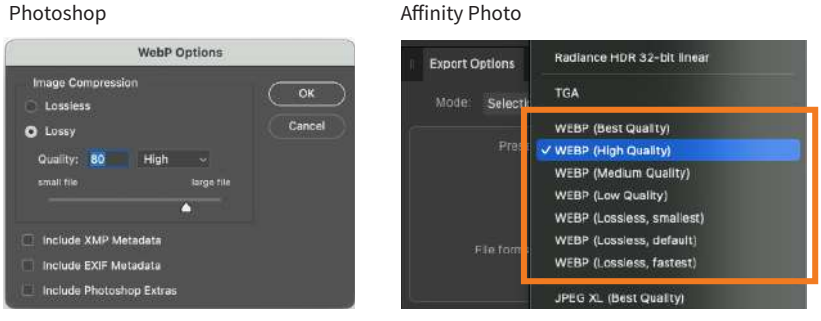


FIGURE 27-9. WebP settings in Photoshop (left) and Affinity Photo (right)

WebP's lossy compression works efficiently on areas with soft blends, so applying a slight blur to the image (or less important areas of it) can wring a few extra KB out of the file size. For example, when exporting the painted portrait in FIGURE 27-4 as a WebP, applying the blur technique reduced the file size from 14 KB to 11 KB.

Optimization Tools

Always run your images through an optimization tool as the last step in the image production process.

Even if you choose the most appropriate format and take full advantage of all the settings in your image editing tool, there's a good chance that you can squeeze down the file size of your images even further using an optimization tool.

Fortunately, there is no shortage of free drag-and-drop image optimization tools online. Some are lossless, meaning they do not alter the appearance of your image, but rather find the file savings by tossing out chunks of code dedicated to metadata, color profiles, and other information. Others compress the image further with lossy compression. Some allow bulk uploads; others restrict you to one image at a time. They also differ in what file formats they accept, the amount of control they give you over the compression, and

the maximum file size. A few can convert from one format to another. You will also find that some of these tools offer paid "pro" packages with more options and higher maximum limits.

I've rounded up a few free optimization tools for you here. These are all good options if you don't have too many images to process, and a quick web search for "best web image optimization tools" will turn up *plenty* more, so you should be able to find one that suits your needs:

Every optimization tool works differently. It is recommended to have a few options available.

- **Squoosh** (*squoosh.app*), designed by the Chrome Labs team, accepts many file formats (including WebP and AVIF) and gives you full control over the optimization settings. The downside is that you can only work on one image at a time.
- **Optimizilla** (*imagecompressor.com*) can optimize JPEG, PNG, and GIF images and allows you to compress up to 20 files at once. You can control the JPEG quality level and check the resulting image against the original in a preview window with a slider.
- **Compressor** (*compressor.io*) allows you to choose whether you want lossless or lossy compression and provides some custom quality settings as well. You can upload multiple images at once in JPEG, PNG, WebP, GIF, and SVG formats. The Compare button reveals before and after previews with a slider.
- **ImageOptim** (*imageoptim.com*) is an easy-to-use tool that's available as a free drag-and-drop application for macOS or as a paid web service. It makes files smaller by running them through several image optimization tools and combining their results. It also removes metadata and has a lossy compression option.

If you are working on a large site with lots of images and you are using a build tool or task runner, check for plug-ins that make image optimization part of the overall development workflow. Content management systems (CMSs) such as WordPress and Drupal have image optimization plug-ins as well. Some compress your images further, and some automatically convert the images to WebP or AVIF on your behalf. If you use a CMS, look into what image optimization tools are available. Later in this chapter, we'll look at image automation services that handle all your image needs, including optimization, for you.

Now you have some strategies for making your images as small as possible, including applying techniques that are adapted to the file format during the creation process and using image optimization tools to squeeze them down even further after they are made. Let's put them to the test in EXERCISE 27-1.

EXERCISE 27-1. Optimize some images

In this exercise, we'll see how small we can make the files for our croissants photo and ad banner. I'll be using Photoshop's "Save for Web" tool as well as the free online Squoosh tool, which offers similar settings (and more!). If you'd like to follow along with either of these tools, the starter images (*croissants.jpg* and *ad-banner.png*) are available with the materials for this chapter.

Optimizing a JPEG

We'll begin by optimizing the croissants JPEG image by using the Squoosh online optimizer. Go to ***squoosh.app*** in your favorite browser and drag the *croissants.jpg* file onto the page. (Note that the starter image is the full-quality JPEG, coming in at 299 KB). Squoosh pops up a split preview showing the before and after versions with a slider (FIGURE 27-10). On the right, choose MozJPEG from the Compress section. You can then fine-tune the Quality setting while keeping an eye on the preview quality and resulting file size. Under "Advanced settings," you can select "Progressive rendering" to make the image display in a series of passes while also slightly reducing the file size, and you can use the Smoothing slider to apply a slight blur to the whole image to make the JPEG compression work more efficiently.

Try out a few settings and pay attention to how they affect the image quality (zooming in on the preview can be helpful). I've decided that the loss of quality is barely noticeable at a Quality setting of 75 (honestly, I don't mind the results even at 50). I'm opting for progressive rendering because it's a pretty large image, but I don't like the effect of smoothing, so I'm leaving that at 0. With these settings, I was able to reduce the file size by 87% to just 38.6 KB. That's a little smaller than the results I got using Photoshop's Export As feature with a medium quality setting (42 KB).

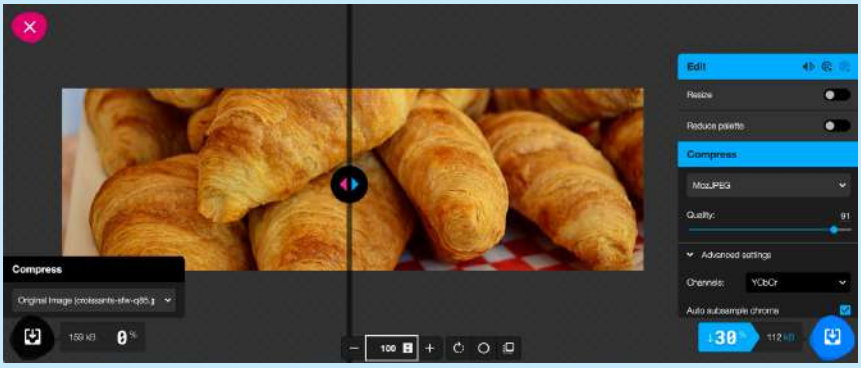


FIGURE 27-10. JPEG optimization settings in Squoosh

I'm going to do the same thing again using Photoshop, for comparison. For additional optimizing tools in Photoshop, use the **File → Export → Save for Web (Legacy)** feature (FIGURE 27-11). This gives you the option to make a progressive JPEG, and I prefer the 0–100 quality scale to the 1–7 scale available with Export As. Keep in mind that these two export methods use different compression algorithms, so 50 is not necessarily the same quality as 4. Another thing to keep in mind is that Adobe is continuing to work on the compression algorithms in the Export As feature, so it may compress more effectively than the legacy option down the line. The "Save for Web" panel shows up to four previews so you can compare image quality at various settings.

EXERCISE 27–1. Continued

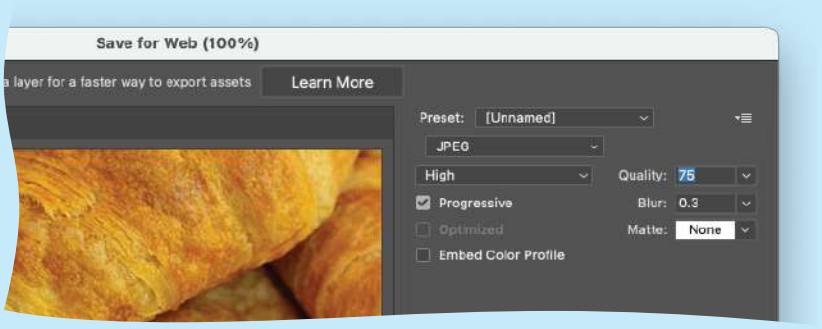


FIGURE 27–11. Photoshop's "Save for Web" feature provides access to a more granular quality setting and the ability to make the JPEG progressive.

Optimizing an indexed color PNG

When experimenting with exporting the *ad-banner.png* file in different formats and quality settings in EXERCISE 25–1, we found that we got the best results when saving the image as an indexed color PNG-8 (my file was 19 KB). Let's see if we can do even better, starting with the Squoosh tool.

Drag the provided file into the ***squoosh.app*** window to access the preview and optimization settings (FIGURE 27–12). Our primary tool for optimizing a PNG-8 is to reduce the number of colors. Under Compress, select Browser PNG from the menu. Above that, turn on the switch next to "Reduce palette," and you can use the slider to reduce the number of colors in the indexed color table (or just type in the number). I usually start with 64 colors (6-bit) then check the preview to see how well the image holds up. Because this is a fairly monochromatic image, I think the result is acceptable at 32 colors (5-bit); the muffin image and typography stay intact. You can also control the amount of dithering. Setting it to 0 removes any speckle patterns and slightly reduces the file size, but removing dithering may compromise certain images. It's just another thing to play around with.

The resulting file is 24 KB, which is a decent savings over the original 45 KB file.

FIGURE 27–12. Optimizing a PNG-8 with the Squoosh online optimization tool

EXERCISE 27–1. Continued

OK, one more time, this time using Photoshop's "Save for Web" option. Open the original *ad-banner.jpg* and select **File → Export → Save for Web (Legacy)**. When you choose PNG-8, you get all the settings for optimizing the file (refer back to FIGURE 27–5). Again, I'm impressed by the image quality with just 32 colors (FIGURE 27–13). Note that you can also choose what palette Photoshop uses when reducing the colors from the full RGB spectrum to indexed color (I find Selective gives me the best results, but feel free to experiment). You can also turn off dithering altogether or select the amount of dithering in percentage values, which will affect the size of the resulting file. I'm going with the highly subjective value of 80%, which cuts out some dithering, but allows it to do its thing in the muffin image and around the typography. (You can ignore the Web Snap option, which is an outdated setting based on the obsolete "web-safe" palette.)

The resulting file using "Save for Web" comes in at a tiny 11 KB! Why are the results so different from Squoosh's? I have no idea. It just goes to show that all optimization tools are not equal. It's good to try a few different options if you're serious about reducing image file size.

PNG with 256 colors

PNG with 32 colors

FIGURE 27-13. For this particular image, both the photographic areas and the typography show little sign of degradation when the colors are reduced from 256 to 32.

The 11 KB result is good, but I wondered if I could make it even smaller, so I dragged my new 32-color PNG into the free online Compressor tool (*compressor.io*). I selected lossless compression so the image would stay the same, and what do you know...the resulting image was only 9 KB (18% smaller)!

As you can see, manually optimizing images and running them through online tools is well worth the effort—I took the ad banner PNG from 45 KB to 9 KB, while maintaining acceptable image quality. You can probably also see that doing this individually for each of your images could get a little cumbersome, so if you have a lot of images to produce, consider automating the process, as we'll discuss in the **"Automate It!"** section.

RESPONSIVE IMAGE SETS

One of the questions in the image strategy flowchart in FIGURE 27-1 was whether the images need to be responsive. As you will recall, using responsive images refers to the practice of creating multiple versions of the same image, providing information about the available images in the markup, then allowing the browser to choose and download only the most appropriate file (see **Note**). It's about performance! The goal is to avoid transferring unnecessary image data, which also speeds up page display and improves the user experience. Why serve an image thousands of pixels wide to a user with a 400-pixel-wide smartphone when there's an alternative?! And conversely, why make a user with a high-density desktop monitor suffer a blurry 400-pixel image?

Now, I'm not going to lie…responsive image strategy is a gnarly topic. It's extra work to make multiple images and to craft the proper responsive image markup to deliver them. For sites with a lot of images, it makes sense to automate the process rather than to produce all of the images and write complex markup manually. But if you are working on a small personal or business site with a manageable number of images, and you want to optimize performance as much as possible, then you might want to dive deeper into responsive image techniques.

I'm going to focus on producing the set of images in this chapter. The detailed markup instructions can be found in my article "Responsive Image Markup" at *learningwebdesign.com/articles/*. I will also provide links to articles that I've found useful throughout this section. In the meantime, knowing a little about the required markup will give you the mental model required to make better sense of the image asset production strategies I'll be introducing here. The **"Just Enough Responsive Image Code"** sidebar should give you the background information you need.

NOTE

Because SVGs can be resized without loss of quality, only one image file is required when you use that format. However, there are still tips for using SVGs in responsive layouts that you should be aware of; I'll cover these in **Chapter 28**.

How Many Images Do You Need?

It's clear that there are benefits to providing a set of images in a range of sizes to avoid unnecessary data downloads. But how many and what sizes do you need to provide? *Nobody knows!* OK, I'm joking, but barely. Because of the seemingly endless combinations of image sizes and device pixel densities, and because browsers vary in how they pick from the available images, there are no clear-cut instructions for what sizes are needed. What we do have are some general strategies suggested by experts in the field, as well as a couple of tools that make the process easier. Here are a few approaches, some requiring that you have a working prototype of your page (even better if it has real content).

Just Enough Responsive Image Code

Although I won't go deep into responsive image markup, I think it will be useful for you to see some examples to provide some context for the image production strategies discussed in this section. The responsive use cases for images embedded with HTML are based on resolution, device pixel ratio (DPR), and art direction. You can also make a CSS background image responsive with media queries.

Resolution-based responsive images

For a browser to download only the most appropriate image from a set, it needs to know:

- What images are available and their actual (intrinsic) pixel widths
- What size the **img** element will appear at in the layout

Equipped with this information, it looks at the size of the viewport and the device pixel ratio (DPR) of the screen, picks the image that fits the space in the layout most efficiently, and downloads only that file.

We can provide all of those details in the **img** element. The **srcset** attribute specifies a comma-separated list of the available images. For each image, you indicate the image width in pixels with a **width descriptor** (indicated by a **w**). The **sizes** attribute tells the browser what size the image element will occupy in the layout at various viewport widths. Its value is a list of media conditions (like you've seen in media queries) that specify the width of the **img** when each condition is met, ending with a default width. The **src** attribute provides a fallback image for non-supporting browsers.

In the following example, I've provided three JPEGs of the same image at widths of 400, 750, and 1,500 pixels. The **sizes** attribute tells the browser that when the viewport is 800 pixels or wider, the image will appear at exactly 750 pixels wide. Otherwise, it will occupy 100% of the viewport width.

```
<img src="strawberries-750.jpg" alt="baskets of
ripe strawberries"
     srcset="strawberries-small.jpg 400w,
             strawberries-medium.jpg 750w,
             strawberries-large.jpg 1500w"
     sizes="(min-width: 800px) 750px, 100vw">
```

Using **srcset** with **w** descriptors is by far the most common responsive method because image sizes tend to be flexible in responsive layouts. However, if you have an image that will stay at the same dimensions in the layout and you want to swap it out based solely on the DPR of the screen, you can instead use a **density** (**x**) **descriptor** to provide the DPR and its respective image file. The **sizes** attribute is not necessary in this scenario. In this example, I've indicated that there is a file (*bunnies-@1x.jpg*) appropriate for screens with standard resolution and another, larger image (*bunnies-@2x.jpg*) created to look crisp on 2x displays:

```
<img src="bunnies-@1x.jpg" alt="a basket containing
three bunnies"
     width: 300px;
     height: 400px;
     srcset="bunnies-@1x.jpg 1x,
             bunnies-@2x.jpg 2x">
```

The art direction scenario

Another responsive approach is the art direction scenario, in which you provide images with different content based on screen size. For example, you may want to crop into an image or remove text overlays when the image appears on small screens. This option uses the **picture** element to tell the browser exactly which image you want it to use in certain conditions. The **img** element is required. In this example, the **source** element is used to provide images aimed at different device widths:

```
<picture>
  <source media="(min-width: 1024px)"
          srcset="icecream-large.jpg">
  <source media="(min-width: 760px)"
          srcset="icecream-medium.jpg">
  <img src="icecream-small.jpg" alt="hand holding
ice cream cone and text that reads Savor the
Summer">
</picture>
```

Responsive background images

Note that the previous solutions are all for images that are placed in the flow of content with the **img** element in the HTML file. It is common for CSS background images to resize based on the layout as well, so it makes sense to create a set of these images at a range of sizes. Then, use media queries in the style sheet to base the image source on the size of the viewport or container. Demián Renzulli's article "Optimize CSS Background Images with Media Queries" (*oreil.ly/SqETB*) provides clear examples.

Largest, smallest, and best guess in between

At the very least, you'll want to provide images that work at the largest and smallest sizes the image will appear at in the layout, and another set at twice those dimensions for 2x displays (see the **"Prioritize 2x Displays"** sidebar). It's determining how many sizes you need in between the endpoints that gets tricky. If the range isn't that large, you may find that providing small, medium, and large versions is fine. If there is a big difference, images for more breakpoints may be required. Conversely, if there is very little difference between the endpoints, providing one image size (and its 2x version) may suffice.

Here's the thing to keep in mind: the browser doesn't know your logic for creating images; it just picks from all the images available. So, although you may provide an 800-pixel image with the intent of it appearing on large displays at 1x, the browser may also think it is the best option when the image is 375 pixels wide on a 2x display. The takeaway is that you don't necessarily need to supply images sized precisely for every breakpoint at multiple scales. What is important is to provide a good distribution over the range of pixel dimensions and file sizes, without sinking too much time into image production.

If that's too loosey-goosey for you, here is an approach that uses a tool for more precision.

Use the RespImageLint bookmarklet

If you have a working prototype of your page (including `img` elements that have already been marked up with `srcset` and `sizes`), you can use the RespImageLint bookmarklet created by Martin Auswöger to generate the optimum `sizes` value for you. It works by running a script that resizes your page behind the scenes and calculates the actual width of the image in a range of viewport widths. The new `sizes` value it generates can be used to replace your initial value and acts as a guide for generating the images at in-between sizes that make mathematical sense.

Get the RespImageLint bookmarklet at ***ausi.github.io/respimagelint/*** (see **Note**). To use it, drag the bookmarklet to your browser's bookmarks bar. Open the page you want to analyze and click the bookmarklet, and it will return a report on all the images on the page.

Masa Kudamatsu provides an excellent tutorial on using RespImageLint in a simplified responsive image workflow in his article "Responsive Images: DIY Implementation in 6 Steps" (***oreil.ly/80DCp***).

Generate images based on file size

If your image is used at a wide range of sizes, providing a range of selections based on *file size*, not pixel dimensions, may be a more appropriate approach. You can trust the browser to make the right selection, scaling up or down slightly as required.

Prioritize 2x Displays

In **Chapter 26**, you learned that the screens of devices of all types now have device pixel ratios of 2x, 3x, and even 4x. The vast majority of smartphones have DPRs of 2x or higher, and the percentage of laptop and desktop monitors with high-density displays is growing. For that reason, it is time well invested to make sure that important photos and graphics look crisp at higher screen resolutions by providing images with larger pixel dimensions (see **FIGURE 25–19**).

However, there is evidence that for photographic images, the human eye cannot detect the quality difference between 2x and 3x, so it generally isn't worth the extra bytes to create a version of the image at 3x or higher. In fact, some designers find that images created at 1.5x look good enough on 2x displays and saves data. For UI elements, logos, and other small images with crisp edges, you might decide that a 3x or 4x version helps ensure the image looks as sharp as the surrounding text, but then again, those types of images are better handled with vector-based SVGs.

The upshot is that when creating image sets for a responsive image, it is recommended that you provide a version for 2x displays but probably not any larger.

NOTE

*If you run into problems with the bookmarklet, there is a Chrome extension by Peter Neumann that provides the same functionality that you can try (it even uses the same code). Go to **chromewebstore.com** and search for "Responsive Image Linter."*

In the file size approach to breakpoints, you create a set of images with file sizes that step up in fixed increments, such as 20 KB or 40 KB. Granted, that takes a lot of extra work and may not be feasible to do manually for a site with a lot of images.

Fortunately, there is a tool that generates the images for you. The Responsive Image Breakpoints Generator by Cloudinary (*responsivebreakpoints.com*) lets you upload a large image and set the maximum/minimum dimensions, the size step, and the maximum number of images, and it generates all the images automatically. FIGURE 27-14 shows the tool used to create a set of strawberry images at 20 KB increments.

For more information about the file size–based approach, I recommend these articles:

- "Introducing Intelligent Responsive Breakpoints Solutions" by Nadav Soferman (*oreil.ly/uZ0lO*). This is Cloudinary's own explanation of its Breakpoints Generator tool.
- "Responsive Image Breakpoints Generator, A New Open Source Tool" by Nadav Soferman at *Smashing Magazine* (*oreil.ly/KN0yd*). This article introduces the tool mentioned in this section and provides a lot of background information about the approach.
- "Responsive Images 101, Part 9: Image Breakpoints" by Jason Grigsby (*oreil.ly/n-54l*). Jason's whole Responsive Image 101 series is a wonderful resource.

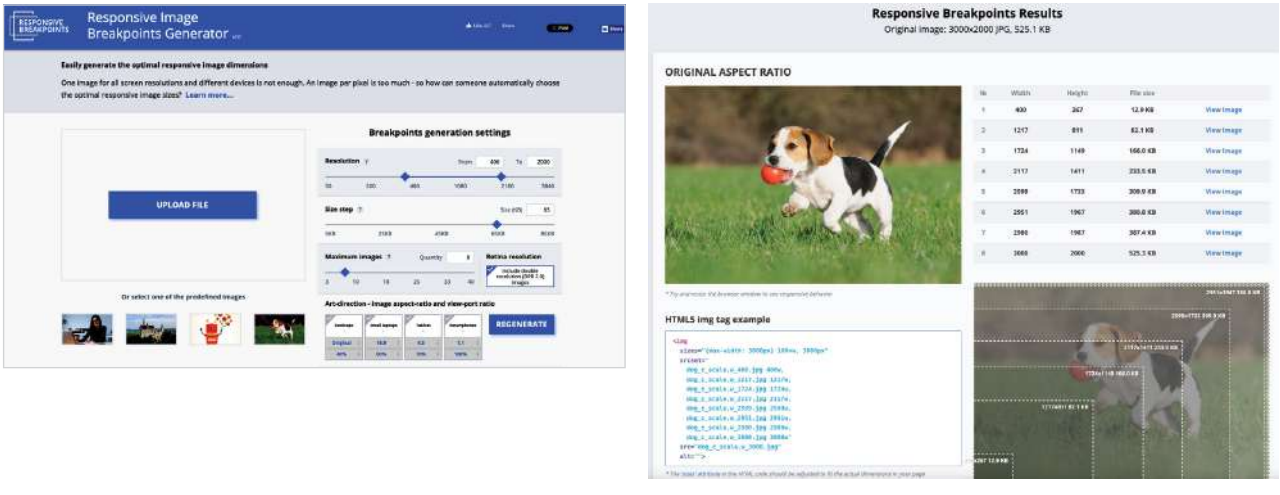


FIGURE 27-14. The Responsive Image Breakpoints Generator by Cloudinary generates image files for you.

TOOLS, TIPS, AND TECHNIQUES

Now that you have the big picture on image production strategy, let's look at some methods for ensuring your images are created at the best quality.

Use Vectors Whenever Possible

One way to make images as sharp as possible on high-density screens is to use vector-based images saved in SVG format. Because the vector images get rendered on the fly, they will stay perfectly sharp at all sizes and resolutions (no responsive markup required!).

In fact, even if you anticipate saving design elements in bitmap formats like PNG, starting your design work in a vector-based program such as Figma, Sketch, Adobe Illustrator, or Affinity Designer gives you the most flexibility when exporting high-resolution versions. You can even export several sizes aimed at various pixel densities at once, as we'll discuss soon. If Photoshop is your page layout design tool of choice, you can take advantage of its vector tools, such as shapes, paths, and imported vector Smart Objects for common web page elements like buttons, icons, and illustrations. The vector source can be exported to any pixel dimensions, whereas elements designed in pixels get fuzzy when they are **upscaled** (resized larger). Which brings me to the next tip...

Image Vectorization

It is possible to convert simple bitmap illustrations to vectors. There are several online image vectorization tools that trace an image and save it as an SVG. One free tool is Vectorizer (*vectorizer.com*), but a web search will turn up others, many charging a fee. If you have Adobe Illustrator, you can use its Image Trace tool (**Object → Image Trace**) to trace the lines of a placed bitmap image. The quality of the resulting vector image depends on the tool and the complexity of the image you feed it, but it's certainly something worth trying if you want to break your graphics free from a specific image resolution.

Start with a High-Quality Original

To preserve the image quality of photographs (and other necessarily bitmap page elements) at large scales, start with an image source that is at least as large as your largest scale. For example, if you know that the 2x version of an image is 2,160 pixels wide, your source image should be that wide or wider. It is always preferable to make a digital image smaller (**downscale**) than to make it larger. (That said, I have some tips for upscaling in the upcoming **"Upscale with AI"** section.)

Resizing images in the export process is a fine option. FIGURE 27-15 shows resizing options in Photoshop's Export As dialog box, but you will find similar settings in other programs. If you need to produce a set of responsive images, resizing during the export process has the advantage of efficiency—just enter the desired image dimensions in the dialog box and repeat. However, the convenience you gain is paid for with some loss of control over the image quality.

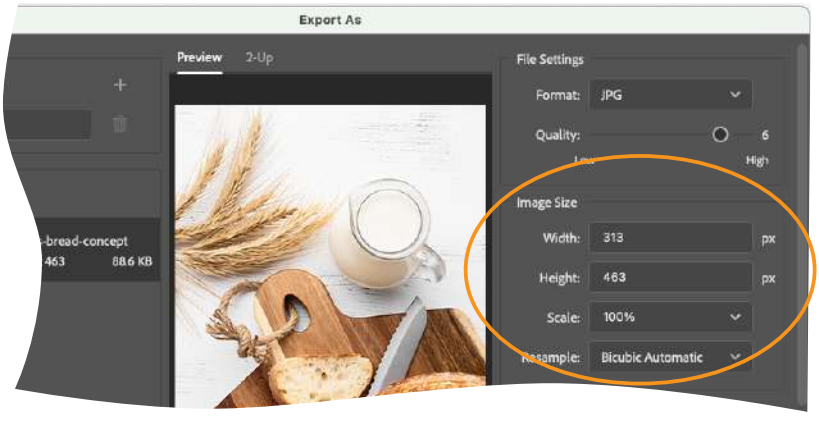


FIGURE 27-15. Resizing images manually (as shown here in Photoshop's Export As dialog box) is an option if you have a manageable number of images to produce.

If you want more control over the resulting image, you can resize the image to the target dimensions manually before exporting using Image → Image Size (see **Warning**). This method gives you an opportunity to make adjustments to the image, such as sharpening it up, before committing to the export.

Resizing an extremely large photographic image to a small size appropriate for a web page layout can blur the image if you do it in one step, such as on export. I find I get better results if I reduce the dimensions a little bit at a time, sharpening the image after each pass. For example, if my source image was 4,000 pixels wide and I needed to get it down to 250 pixels, I'd start by resizing it manually by 50%, then use the Sharpen tool to sharpen it up again. I'd repeat the steps until I got the image to its target size. It's definitely more work, but it's worth it if I am happier with the results.

WARNING

Be very careful not to save the document, or you'll replace your large original with something smaller. Working on a copy of the original file is good insurance against destroying the source image.

Embed Large-Scale Bitmaps

Keep high-quality originals in mind when designing the whole page or interface as well. Most design programs allow you to export single assets from the page, but you need to be mindful of the export resolution.

When designing layouts in a bitmap image editing tool like Photoshop, GIMP, or Affinity Photo, it is common to work at the default 72 ppi (1x) scale. Unfortunately, images exported from a 1x layout design may suffer when upscaled to 2x or 3x on export. The trick to avoiding this problem is to embed the high-resolution version of image in the design file.

In Photoshop, to take advantage of the full image resolution, add the image to your design as a linked Smart Object by choosing File → Place Linked, then resize it to fit your design. The Smart Object is like a placeholder for

the image in your 1x design. When it comes time to export at various scales, Photoshop references the external high-resolution version, and you end up with full-resolution exports.

In vector-based design tools like Illustrator, Figma, Sketch, and Affinity Designer, "place" the high-resolution source image on the artboard and resize it to fit the needs of the layout. This gives the program all the pixel information it needs to export high-quality, large-scale assets when it is time to do so.

Export Multiple Scales at Once

As you've learned, it is common to produce multiple versions of an image aimed at devices with higher pixel densities (1.5x, 2x, etc.). Thankfully, many image tools have features that make it easy to export multiple scaled versions at once. It's a nice little time (and math!) saver.

Keep in mind that the image you export will be doubled or tripled in size on export, which should raise a red flag. For this feature to work its best when working at 1x scale, you'll want to start with vector-based elements and/or high-resolution images embedded in the design file, as discussed previously.

Look for the option to add scaled images wherever your tool handles exports. FIGURE 27-16 shows the scale option interfaces for a number of popular design tools. Clicking the + symbol adds another exported image; simply select the scale from a pop-up menu. When you click Export, all of the versions will be created at once and named with suffixes indicating the scale (-@*2x*, -@*3x*, etc.).

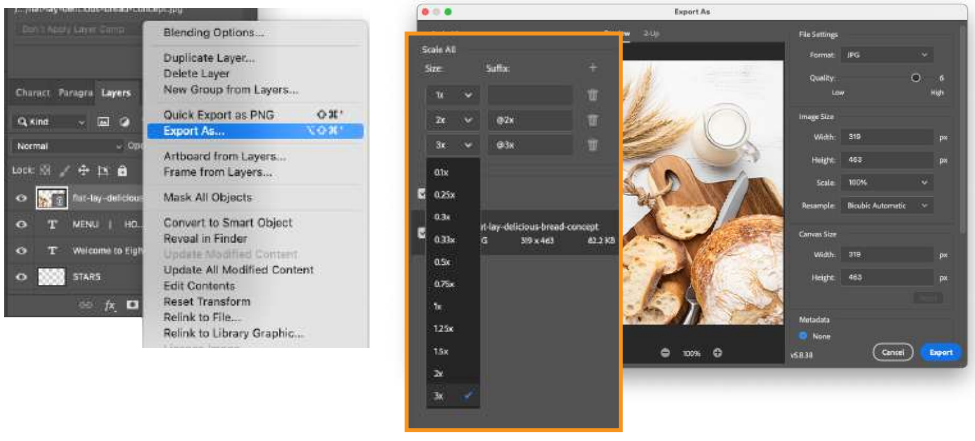
Upscale with AI

Starting with a big original image is the ideal, but in reality, sometimes our source images just aren't as large as we need. In these cases, upscaling is a necessity. Fortunately, thanks to AI, it's now possible to enlarge an image while retaining sharp detail.

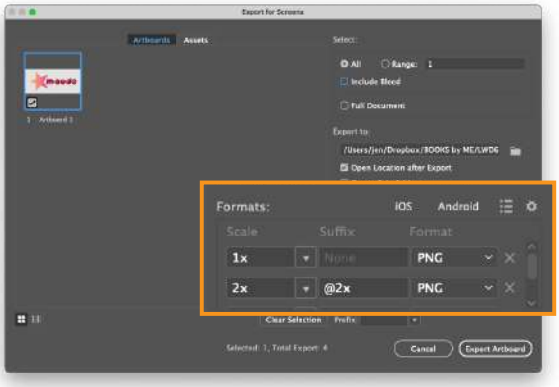
When you upscale an image, the image editor needs to fill in the added pixels. In the traditional image resizing process, our only tool for controlling how those pixels were generated (and therefore the quality of the image) was the resampling method. In Photoshop, when you select Image → Image Size or File → Export → Export As, you can choose from a number of resampling methods (FIGURE 27-17). When enlarging an image, the recommended methods are Bicubic Smoother, Bicubic Automatic (allows the software to choose for you), or Preserve Details. You may have to try each of them to see if one is a clear winner for your image. You may also need to sharpen the image slightly after it is enlarged.

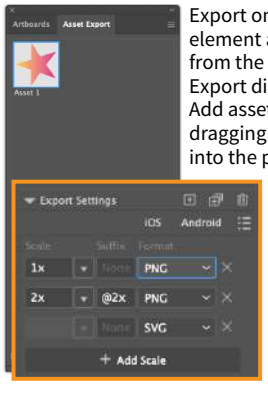
Photoshop
Right-click a layer to export one layer element at a time.

Select the file format under File Settings (right). Add scales in the Scale All section (left).



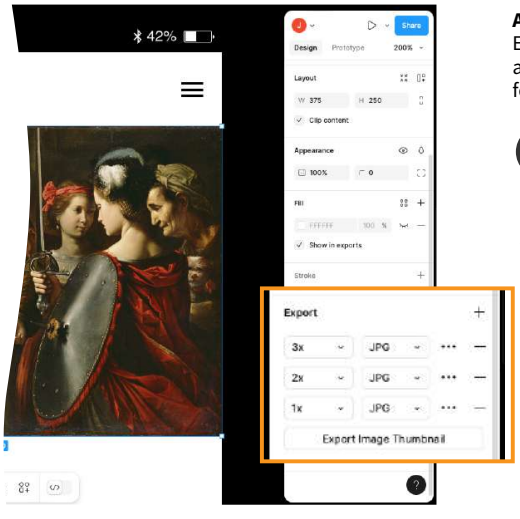
Illustrator
Choose "Export for Screens" from the File menu to export the entire artboard. Choose the file format and add scales under Formats.





Export one element at a time from the Asset Export dialog box. Add assets by dragging them into the panel.

Figma
Select the item to export and add scales in the Export panel.



Affinity Designer
Enter the Export Persona by clicking its icon. Click the arrow icon next to the slice you want to export to reveal format and scale settings.

Export Persona icon

FIGURE 27-16. Newer design tools allow you to export multiple high-density sizes at once.

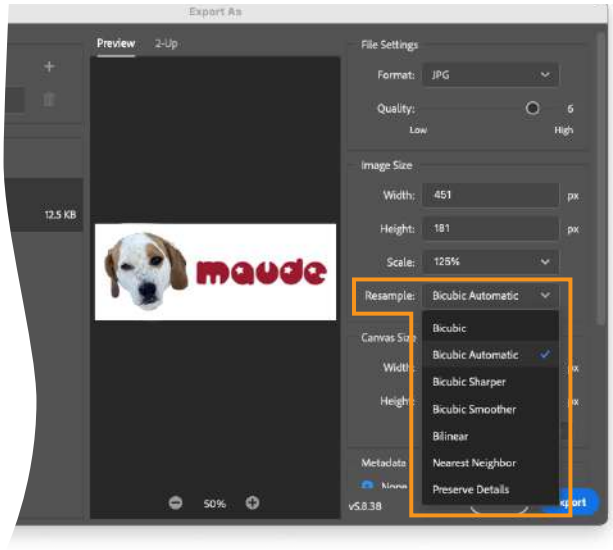


FIGURE 27-17. Resampling method options in Photoshop

These days, we can leverage AI's strength at generating images to fill in those extra pixels in a more meaningful and accurate way. In Photoshop, the Super Resolution option (accessible through the Camera Raw plug-in or in the Lightroom app) doubles the width and height of an image and produces truly impressive results. Here are Adobe's instructions for how to use it:

1. Open it: From Camera Raw (the plug-in, not the Photoshop filter), open your original image.
2. Enhance it: In the filmstrip, right-click or control-click an image file and select Enhance from the menu.
3. Super Resolution it: Check the box that reads Super Resolution. You'll see a time estimate beneath it.
4. Save it: Click Enhance. The algorithm will do its work, and the new, larger image will be automatically produced, with the file extension DNG attached to it.

Photoshop also has a Super Zoom filter (available in the Filter → Neural Filters menu) that uses Adobe's Sensei AI to blow up the image, but I find the results less realistic than Super Resolution's. FIGURE 27-18 shows a detail of an image enlarged 200% using each method. In the traditional upscaled image on the left, you get a lot of jagged and highlighted edges. Super Resolution does a nice job of filling in the wood grain while creating a smoother edge on the knife. The Super Zoom version is the smoothest, which looks nice in the wood grain, but the loaf of bread looks like it was painted with watercolors (a telltale sign of AI as of this writing). Because these examples are hard to compare in print, I've included copies of each of these files with the materials for this chapter.

Image Size
(200% with Bicubic Smoothing)

Super Resolution
(in Camera Raw plug-in)

Super Zoom
(in Neural Filters)

FIGURE 27-18. A comparison of Photoshop's upscaling methods

If you don't use Photoshop, no worries! There are plenty of AI-assisted image upscalers available online (see **Note**). Many of them have free plans in addition to paid subscriptions, such as LetsEnhance (***letsenhance.io***) and ImgLarger's AI Image Enlarger (***imglarger.com***). If you think you will be doing a lot of upscaling and want high-quality results, another option is to invest in dedicated software like Topaz Labs' Gigapixel (***topazlabs.com***). This landscape is changing rapidly, so I recommend doing your own "AI upscalers" search for up-to-date results.

NOTE

Just a personal opinion here: as I write this in 2025, I find that the AI-upscaled images have a distinctive "shimmery" AI look that isn't photorealistic. But AI image generation is evolving rapidly, so I fully expect it to improve.

AUTOMATE IT!

You're probably thinking that this responsive image set production and optimization business is a lot of work. You're not wrong. I mentioned this earlier, but it bears repeating: if your site is image heavy, consider using an image content delivery network (CDN) service that automates the process of responsive image generation. Actually, consider it even if you have a small site (many services are free). As Jason Grigsby says in the article I referenced earlier, "Humans shouldn't be doing this." I couldn't agree more (unless you have a penchant for repetitive tasks or only have a few images to bang out). Before we get into image CDNs, let's take a brief look at CDNs in general

Content Delivery Networks

A **content delivery network** is a globally distributed network of servers that cache your content, so it has less distance to travel to get to your users. The advantages are increased page speed (less distance means faster data transfers) and redundancy, which makes your site less vulnerable to downtime or attacks. That gives you the general idea, but for a more thorough explanation

of how CDNs work, see "What Is a Content Delivery Network (CDN)—A Beginner's Guide" by Rahul Nanwani of ImageKit (*oreil.ly/i6Yc0*).

Image CDNs

Image CDNs provide specialized image optimization, manipulation, and delivery services over a CDN. They offer all the advantages of a distributed network, while also handling the images (and videos) for you. In a nutshell, you upload one image at the highest quality, and the image CDN software creates a version of the image on the fly that has the optimum combination of file format, image dimensions, and file size for the user's browsing environment. That image may be stored so it will be available to the next browser that requests it even faster.

You don't need to be a big enterprise to take advantage of image automation.

In addition to uploading the image, you may need to write the URL for the image in a particular way, according to your CDN's requirements. Rahul Nanwani has a nice explainer on image CDNs on the *ImageKit* blog (*oreil.ly/ENKTP*).

Image CDN companies are working on image optimization constantly, so they've got fancy algorithms that take advantage of the latest and greatest in image technology. They can even use AI to crop images in meaningful ways to accommodate the art direction responsive scenario. I'm only scratching the surface of all that image CDNs have to offer. If you anticipate working with images (especially responsive or user-uploaded images), these services are well worth researching.

There are *many* image CDNs out there, but some of the most popular as of this writing include Cloudinary (*cloudinary.com*), ImageKit (*imagekit.io*), imgix (*imgix.com*), and Kraken (*kraken.io*). Cloudinary, ImageKit, and imgix offer "free forever" plans that are sufficient for smaller-scale sites, so you don't need to be a big enterprise to take advantage of image automation.

Other Automation Options

There are a few other ways to automate the image optimization and delivery process that are worth mentioning:

Self-hosted image CDNs

If you are working with an engineering team that can handle setup and ongoing management, you can run image manipulation software on your own servers. Many image CDN services, including the ones I mentioned, also offer self-hosted versions of their software. Or you could use Thumbor (*thumbor.org*), which is an open source, self-hosted-only solution.

Content management system image optimization

If you use a CMS like WordPress or Drupal, you can take advantage of image plug-ins that handle optimization for you.

Build tools

Site build tools can be configured to use scripts and command-line tools that automate image optimization as part of the build process. These are different from image CDNs in that they help create a set of static images that get uploaded to the server and do not generate images on the fly. The popular, open source ImageMagick (*imagemagick.org*) is one such tool.

WRAPPING UP IMAGE PRODUCTION

And with that, we end our tour of image production techniques. We covered a lot of territory in this chapter, starting with a general strategy for approaching image asset creation. First and foremost, you should have a good optimization toolkit for making image files as small as possible while meeting your image quality goals (the primary image production skill, in my opinion). We also looked at some approaches for creating responsive image sets and tips for producing images with the best quality. And if all that feels too time intensive, you also have a head start on finding services to resize and optimize them for you.

As usual, this chapter ends with a quiz so you can put your new knowledge to work.

TEST YOURSELF

Are you an image asset master? Answer these questions to find out. Answers are in **Appendix A**.

1. Name two alternatives to bitmap images for simple shapes and illustrations.

2. What do responsive images require in terms of image asset production?

3. Which of the following techniques could be used to optimize a JPEG?
   a. Lower the quality setting
   b. Blur the image slightly
   c. Reduce the number of colors

d. a and b

e. All of the above

4. Which of the following techniques could be used to optimize an 8-bit PNG or GIF?

   a. Reduce the amount of dithering

   b. Blur the image slightly

   c. Reduce the number of colors

   d. a and c

   e. All of the above

5. Which of the following techniques could be used to optimize a lossy WebP?

   a. Lower the quality setting

   b. Blur the image slightly

   c. Run it through an optimization tool

   d. a and b

   e. All of the above

6. How can optimization tools make files smaller without affecting the quality of the image?

7. What two pieces of information does the browser need to pick the most appropriate image from a responsive image set?

8. What `img` attribute provides the list of available image files for a responsive image?

9. Why might you use an image CDN?

   a. To create appropriately sized images on the fly

   b. To make image data download more quickly

   c. To improve the security of your site with multiple backups

   d. To take advantage of the latest optimization technologies

   e. All of the above

10. Why might you want to produce an image at 1x, 2x, and 3x scales?

11. Why is upscaling generally avoided?

CHAPTER

28

SVG

(SCALABLE VECTOR GRAPHICS)

IN THIS CHAPTER

Shapes in SVG

Clipping and masking

Filter effects

Interactivity and animation

SVG tools

Production tips

How to scale SVGs

SVGs have made several guest appearances in this book, but in this chapter, they finally get to be the star.

When rendered in a browser window, an SVG may look like an image in any other format, but it's what's happening behind the scenes that makes it truly unique and versatile. First, as the name says, this is a vector format, meaning shapes within SVGs are defined by coordinates and lines, not grids of pixels. That's what makes them scalable—they can resize infinitely without loss of quality.

FIGURE 28-1 shows the same tiger image saved as an SVG and a PNG. The SVG can scale very large without any change in quality. The lines and text stay sharp, regardless of whether the image is viewed at 100 pixels or 10,000 pixels—try doing that with a bitmap image! Now that our web pages and interfaces must work on devices of all scales, from smartphones to high-density monitors and large-screen televisions, the ability to create a single image that looks great in all contexts is an epic win.

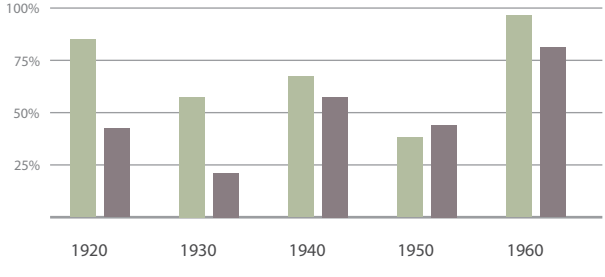
The vector nature of SVG makes it a good choice for icons, logos, charts, and other line drawings (FIGURE 28-2). And because these types of drawings are made up of shapes and paths, the file size is often smaller than for the same image made up of a grid of pixels.

SVGs may contain raster image content as well—in fact, you can do some pretty cool effects with them—but you miss out on the file size savings. SVGs are also an attractive choice for adding animation and interactivity to an interface. We'll review all of these capabilities in this chapter.

Scalability and reduced file sizes make SVG a great format for icons and simple illustrations.

FIGURE 28-1. Vector-based SVG images scale without loss of quality.

The Noun Project



"ben" , Open Clip Art

Ozer Kavak, Open Clip Art

Ghostscript tiger

FIGURE 28-2. The SVG format is appropriate for line-style illustrations.

DRAWING WITH XML

That takes care of the "scalable" and "vector" aspects of SVG. What really sets SVG apart from other formats, though, is that it is an XML language for describing two-dimensional graphics, including shapes, paths, text, and even special filter effects (see the sidebar **"A Quick Introduction to XML"**). Bitmap images are stored as largely unintelligible code (should you care to peek inside), but SVG images are generated by text files that are fairly human-readable. In fact, it is technically possible to create SVG graphics by typing out the code directly in a code editor instead of using an illustration program.

A Quick Introduction to XML

XML (which stands for **eXtensible Markup Language**) is a method for storing and transporting data as text using descriptive markup (tags). XML does not have predefined tags of its own, but rather defines a set of rules for creating custom markup languages. It is a meta-language.

To use a simplified example, if you were publishing or storing recipes, you might use XML to create a custom Recipe Markup Language (RML) that includes the elements `<ingredient>`, `<instruction>`, and `<serving>` to accurately describe the types of information in your recipe documents. Once labeled correctly, that information can be treated as data. In fact, XML has proven to be a powerful tool for sharing data between applications.

Despite the fact that XML was developed with the web in mind, it has had a larger impact outside the web environment because of its data-handling capabilities. There are XML files working behind the scenes in an increasing number of software applications, such as Microsoft Office and Apple iTunes.

XML languages that are used on the web include:

- **XHTML:** HTML rewritten according to the stricter rules of XML
- **MathML:** Used to describe mathematical notation
- **EPUB:** An open source ebook file format
- **RSS** (Really Simple Syndication, or RDF Site Summary): Allows your content to be shared as data and read with RSS feed readers
- **SVG:** The image description language that you'll learn all about in this chapter

XML syntax requirements

Because it is possible for multiple XML languages to appear in the same document, it is important that the syntax be very strict to keep things straight. Shortcuts that are fine in HTML (such as omitting end tags) won't fly in XML languages.

SVG follows the stricter XML syntax, so it is important to follow these code requirements when you're writing SVG:

- Element and attribute names must be lowercase.
- All elements must be closed (terminated), which means they must have a closing tag. To close elements without content (a.k.a. empty/void elements), you add a slash before the closing bracket (for example, `<hr/>`).
- Attribute values must be in quotation marks. Single or double quotation marks are acceptable as long as they are used consistently. Furthermore, there should be no extra whitespace (character spaces or line returns) before or after the attribute value inside the quotation marks.
- All attributes must have explicit attribute values. XML does not support **attribute minimization**, where certain attributes can be reduced to just the attribute value. This is best explained with an example from XHTML, a stricter version of HTML rewritten in XML. In HTML, you can write `checked` to indicate that a form button should be checked when the form loads, but in XHTML you need to explicitly write out `checked="checked"`.
- Elements must be nested properly.
- Special characters must always be represented by character entities (e.g., `&` for the `&` symbol). Note that most HTML named entities do not work in XML; use the numeric Unicode code point references instead.
- Scripts must be contained in a `CDATA` section so they will be treated as simple text characters and not parsed as XML markup. There is an example in the **"Interactivity with JavaScript"** section later in this chapter.

SVG Elements

Just as HTML is a markup language for text-related elements, SVG is a markup language that includes elements found in two-dimensional graphics, such as the following:

- Elements for drawing lines and shapes, such as **`circle`**, **`rect`**, **`ellipse`**, **`path`**, **`line`**, **`polyline`**, and **`polygon`**
- A **`text`** element for adding text content
- Elements for organization, such as **`g`** for grouping shapes together, and **`use`** and **`symbol`** for reusing drawings
- Elements for clipping (**`clipPath`**) and masking (**`mask`**) image areas into interesting shapes
- Elements for raster effects, such as **`linearGradient`** and **`filter`** for Photoshop-like filter effects

NOTE

Lists of all the SVG elements and attributes can be found at w3.org/TR/SVG11/.

Of course, this is by no means an exhaustive list (see **Note**), but it should give you the general idea of what SVG is about. An example will make it even clearer. FIGURE 28-3 shows an SVG image, *simple.svg*, that contains a sampling of simple SVG elements. It's not a masterpiece, I know, but it will introduce you to some common SVG elements.

Here is the content of the *simple.svg* file that generates the image in FIGURE 28-3. If you read through it closely, I think you'll find it's fairly intuitive, but my annotations follow.

```
<?xml version="1.0" encoding="utf-8"?> A
<svg version="1.1" B
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  width="150" height="200" viewBox="0 0 150 200"> C
```

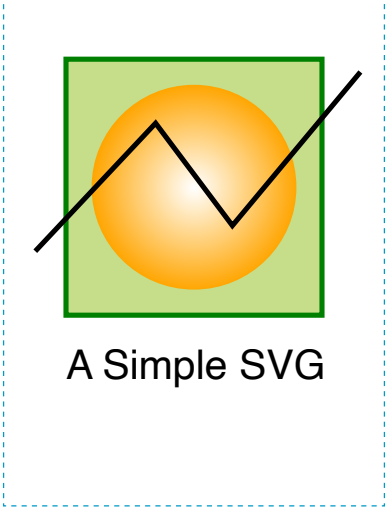


FIGURE 28-3. A basic SVG image, *simple.svg*. The dotted line has been added to indicate the edges of the viewport but is not part of the SVG code.

SVG History and Browser Support

SVG had a long ramp-up over the course of two decades. It was first introduced by the W3C SVG Working Group in 1998, and SVG 1.0 finally achieved Recommendation status in 2001. SVG 1.1 (a more modularized version) was released in 2003, then cleaned up and republished in 2011 (*w3.org/TR/SVG11/*).

The major browsers have supported SVG since 2014, and overall support is at 98% as of this writing (the only exceptions are very old versions of Internet Explorer and Android that most developers no longer accommodate anyway). There may be some inconsistencies in browser support when you get down to the property level and other syntax minutiae, however.

The SVG 2 spec (*w3.org/TR/SVG2/*) was initiated with the aim to be more integrated with HTML5, CSS, and WOFF (Web Open Font Format), but it may be dead in the water. Although Safari supports it completely, most browsers support only a few of the most useful new elements and attributes. If you're curious about the details, see this support chart: *nikosandronikos.github.io/svg2-info/svg2-feature-support/*.

```
<defs> D
  <radialGradient id="fade"> E
    <stop offset="0" stop-color="white"/>
    <stop offset="1" stop-color="orange"/>
  </radialGradient>
</defs>

<g id="greenbox"> F
  <rect x="25" y="25" width="100" height="100" fill="#c6de89"
   stroke-width="2" stroke="green"/> G
  <circle cx="75" cy="75" r="40" fill="url(#fade)"/>
  <path d="M 13 100 L 60 50 L 90 90 L 140 30" stroke="black" stroke-
width="2 fill="none"/> H
</g>

<text x="25" y="150" fill="#000000" font-family="Helvetica" font-
size="16">A Simple SVG</text> I

</svg>
```

Let's take a closer look at the various parts of *simple.svg*:

A. Because this is an XML file, it starts off with some XML business. The first line is an XML declaration that identifies the file as XML. For web SVGs, this declaration is not necessary unless you are using a character encoding other than the default UTF-8, but you are likely to see it in code exported by graphics programs.

B. The entire document is contained in the **svg** root element. Drawing programs generally include the version number (1.1), although it is not necessary. The **xmlns** attribute declares the XML namespace, which tells the browser to interpret this document by using the vocabulary defined in SVG (see **Note**). Namespaces help keep element names straight, especially when there is more than one XML language used in a document. The **xmlns:xlink** attribute allows you to put links and references to external files in the SVG document.

C. The **width** and **height** attributes in the root **svg** element establish the display area (**viewport**), which in this case is 150 pixels wide by 200 pixels tall. The viewport is indicated in FIGURE 28-3 by a dotted line, but this is for illustration purposes only and wouldn't appear in the browser. The default measurement in SVG is the **user unit**, which is the equivalent of 1 pixel, so you don't need to include the "px" unit for the width and height. The **viewBox** attribute controls the size of the drawing itself and enables the image to scale proportionally if you change the dimensions later (see **Note**).

D. Next, we have a **defs** element, which defines elements and effects that will be referenced later in the document by their **id** values. Elements in the **defs** section are created but not immediately rendered. Here I am using **defs** to store a radial gradient pattern, but it could also be used for shapes (like **circle**) or symbols (**symbol**) that get rendered in the document via

NOTE

The **xmlns** *and* **xmlns:xlink** *attributes are not required when the SVG is placed inline in an HTML document.*

NOTE

The viewport and viewbox are discussed in more detail in **"Scaling SVGs"** *later in this chapter.*

the `use` element. Defining a shape, drawing, or effect once and then reusing it is a good way to eliminate redundancy in your SVG code.

E. This `radialGradient` element is made up of two color `stop` elements, one for white and one for orange. It has an `id` of "fade".

F. Finally, we get to the elements that describe the drawing itself. The rectangle (`rect`), circle (`circle`), and path (`path`) that make up the drawing are grouped together with the `g` element and given the name "greenbox". This makes it easy to access those elements as a group in CSS or a script later.

G. The square is created using the `rect` (for "rectangle") element, with its width and height set to 100 pixels. Notice that this and other empty elements are closed (terminated) with a slash (`/`) before the closing bracket, as is required in XML languages.

The `x` and `y` attributes position the rectangle on the pixel grid within the viewport (see FIGURE 28-4 in the **"SVG Coordinates"** sidebar). You can see that attributes are used to provide the dimensions, fill color, stroke width, and stroke color (see **Note**). The center of the `circle` element is positioned with the `cx` and `cy` attributes, and its radius is set with `r`. This `circle` is filled with the radial gradient I defined earlier, which is called by its `id`, "fade", in the `url()` notation.

NOTE

"Stroke" is what SVG calls the line or border around a shape.

H. The crooked line is defined by a `path` element. The `d` (data) attribute provides a series of (*x*, *y*) coordinates that make up the points along the path. All paths start with `M` ("moveto"), which sets the starting position. Each `L` ("lineto") draws a line to the next set of coordinates. Coordinates can be separated by a character space (as shown) or a comma.

I. Finally, we have a bit of text defined with the `text` element. You can see that it is styled with attributes like `font-size` and `font-family`, which should look familiar now that you've learned about CSS. There are many similarities between SVG attributes and CSS styles.

I thought it would be fun for you to get some hands-on experience working with an SVG at the code level, so take some time to put a friendly face on Bobby the Robot in EXERCISE 28-1.

SVG Coordinates

The coordinates in SVGs start at 0 in the top-left corner and increase down and to the right (FIGURE 28-4). The square created with the `rect` element in the example has the coordinates `x="25" y="25"`, which means its top-left corner is positioned 25 pixels from the left edge of the viewport/viewbox and 25 pixels down from the top edge. Some elements, like circles and ellipses, may be positioned based on their center points (`cx` and `cy`).

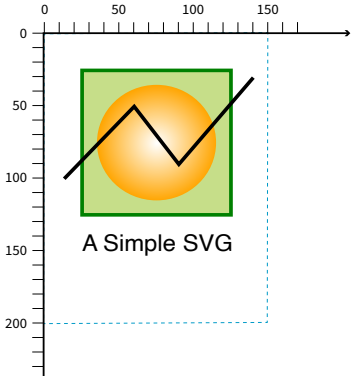


FIGURE 28-4. In SVG, *x* coordinates start on the left and increase to the right, and *y* coordinates start at the top and increase downward. The (0, 0) origin point is at the top-left corner of the viewport.

EXERCISE 28-1. Working with SVG Shapes

In this exercise, you'll write the markup necessary to add eyes, a nose, a mouth, and some text to the Bobby the Robot SVG (FIGURE 28-5).

The starter SVG file, *robot.svg*, is provided with the materials for this chapter. Open the file in the text or code editor of your choice (that's where you'll be doing your work). Also open the SVG file in your browser (that's where you'll see the results of your edits). If you have a large monitor, it might be helpful to arrange the text editor and browser windows side by side. After each step, save your file and refresh the browser, just as you would for an HTML document. The finished SVG, *robot-exercise1.svg*, is also provided with the chapter's materials, so if typing everything out feels daunting, copying and pasting is fine.

Take a look at the starting SVG document and familiarize yourself with it. Its dimensions are set to 300 by 400 pixels, and there are shapes for creating the face, antenna, and so on. The **id** attributes should help you match the elements to each part of the illustration.

1. First, let's give Bobby eyes. Add the following **circle** elements to the document (before the closing **</svg>** tag). The **cx** and **cy** attributes position the center of each circle on (*x*, *y*) coordinates in the user space. The color and stroke are set using SVG attributes:

```
<circle cx="108" cy="230" r="20" fill="#fff" stroke="#000"
  stroke-width="10"/>
<circle cx="190" cy="230" r="20" fill="#fff" stroke="#000"
  stroke-width="10"/>
```

2. Next, let's add a **path** element to give Bobby a little smile. Add the following path, which specifies the (*x*, *y*) coordinates of four points. The **stroke-linecap** attribute makes the ends of the thick rules rounded. (I'll be the first to admit that this is kind of a goofy mouth for a robot, but it does provide the opportunity to create a path without too many points. Feel free to replace it with the shape of your choice.)

```
<path d="M 90 277 L 115 300 L 188 300 L 211 275" stroke="#d00086"
stroke-linecap="round" fill="none" stroke-width="8"/>
```

3. OK, time to give Bobby a nose with this **rect** element:

```
<rect x="140" y="192" width="18" height="76" fill="#377569"/>
```

   That looks fine, but I think a gradient fill would look even better. First, define a linear gradient in a **defs** element at the start of the document. By default, linear gradients are horizontal, so I've used the **gradientTransform** attribute to turn it 90 degrees:

```
<defs>
   <linearGradient id="noseGradient" gradientTransform="rotate(90)">
     <stop offset="0" stop-color="#c3d5d8"/>
     <stop offset="1" stop-color="#377569"/>
   </linearGradient>
</defs>
```

   Now we can use "**noseGradient**" as the fill for the nose:

```
<rect x="140" y="192" width="18" height="76" fill="url(#noseGradient)"/>
```

4. Finally, add a name to this portrait by using the **text** element. Feel free to give your robot any name you please; you don't have to call him (or her) Bobby. You may need to adjust the **x** value to keep the name centered, if that sort of thing matters to you:

```
<text x="110" y="364" fill="teal" font-family="Helvetica"
  font-size="24">BOBBY</text>
```

That's it for now. Hang onto this SVG file for future exercises. Before moving on, however, I encourage you to play around with the values to change colors, strokes, positions, etc. This will give you a feel for how SVG elements are drawn.

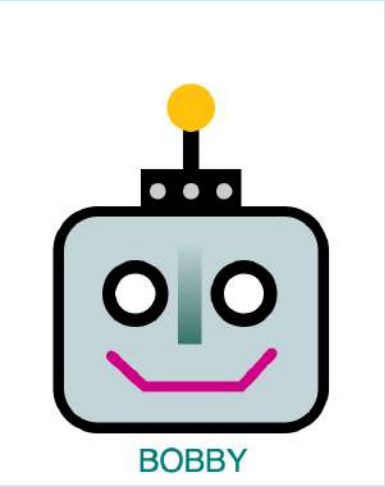


FIGURE 28-5. Bobby the Robot as he'll look at the end of this exercise

Beyond Simple Shapes

SVGs have many other cool image features in addition to simple lines and shapes.

Embedded bitmap images

SVGs aren't limited to vector drawings; you can embed bitmap images in them too (see **Warning**). You might do this in order to apply special effects or add some sort of behavior or interactivity that a PNG or JPEG can't provide on its own. Images are embedded with the **image** element:

```
<image xlink:href="kangaroo.jpg" x="45" y="0" width="100"
height="150"/>
```

Note that because SVG is an XML format, it requires the **xlink:href** attribute to point to the external image file as well as the **xmlns:xlink="http://www.w3.org/1999/xlink"** declaration in the **svg** opening tag.

WARNING

External images won't display for .svg files added to the HTML page with the **img** *element. All external files are blocked for* **img** *element SVGs for security reasons.*

Clipping and masking

SVG lets you selectively show parts of an image and hide others by clipping and masking.

In **clipping**, a vector path is used to "cut out" a section of an image. The parts of the image that fall outside the clipping path are hidden completely. FIGURE 28-6 uses a path in the shape of a star to clip the starry sky image. Clipping paths are defined with the **clipPath** element:

```
<defs>
  <clipPath id="star">
  <polygon points="390,12 440,154 590,157 470,250 513,393 390,307
266,393 310,248 189,157 340,154 390,12" style="fill: none"/>
  </clipPath>
</defs>

<image xlink:href="starrysky_600.jpg" width="600" height="400"
style="clip-path: url(#star)"/>
```

The star-shaped path positioned over the image.

The image is clipped to the path (the dotted border indicates the SVG viewport but is not part of the SVG).

FIGURE 28-6. The star-shaped path is used as a clipping path that reveals part of the underlying image.

Masking works similarly but is a pixel-based effect. Varying levels of darkness in the mask result in varying amounts of transparency at each point, similar to an alpha channel. In SVG masks, pure white areas correspond to 100% opacity, and pure black corresponds to 0% opacity (fully transparent). Levels of gray in between result in different levels of semitransparency.

You can play with the fill color of the mask to reveal more or less of the object it is masking. The effect is more interesting when the mask contains gradients rather than solid fill colors. You can even use an image as a mask (see **Note**).

FIGURE 28-7 shows the same star shape filled with a gradient and used as a mask over the starry sky photograph. Note that the areas of the image that fall outside the mask object are completely transparent, just like with the clipping path.

```
<defs>
  <linearGradient id="blend">
    <stop offset="0%" stop-color="#ffffff"/>
    <stop offset="100%" stop-color="#000000"/>
  </linearGradient>

  <mask id="star" x="0" y="0" width="400" height="381">
    <polygon points="390,12 440,154 590,157 470,250 513,393 390,307
266,393 310,248 189,157 340,154 390,12" style="fill: url(#blend)"/>
  </mask>
</defs>

<image xlink:href="starrysky_600.jpg" width="600" height="400"
style="mask: url(#star);"/>
```

The star-shaped path is filled with a gradient.

The gradient works as a mask in which the light areas allow more of the image to show through. The darker the mask, the lighter the masked image.

FIGURE 28-7. The star shape has a gradient fill that affects the transparency of the masked image.

You can try out embedding an image and giving it a shape with a clipping path in EXERCISE 28-2.

NOTE

If the mask is a color image, it is converted to grayscale based on its luminance (brightness) using a formula that interprets yellows and greens as lighter than reds and blues.

EXERCISE 28-2. Adding an oval background image

We'll be starting with the *robot.svg* file as you left it in EXERCISE 28-1, or you can open the *robot_exercise1.svg* file if you want a fresh start.

1. First, we'll embed the provided *circuits.jpg* image. Because we want it to appear in the background behind the robot illustration, the **image** element must come first (after **defs**). This is because elements later in the SVG document are drawn over elements that precede them. Insert the following **image** element right after the **defs** section:

   ```
   <image x="0" y="0" width="300" height="400" xlink:href="circuits.jpg"/>
   ```

   The **xlink:href** attribute is a special XML-style link required to link to external files. In order to use it, we also need to explicitly define the **xlink** namespace in the **svg** root element. Add the following attribute right after the existing **xmlns** attribute inside the opening **<svg>** tag:

   ```
   xmlns:xlink="http://www.w3.org/1999/xlink"
   ```

   Save the document and open it in your browser. You should now see the circuits photo covering the background of the SVG.

2. Next, let's add a clipping path to give the background image an oval shape. We do that by defining a **clipPath** in the **defs** section, then applying it as a style in the **image** element. Add the following inside the **defs** element:

   ```
   <clipPath id="oval">
     <ellipse cx="150" cy="200" rx="145" ry="195" fill="none"/>
   </clipPath>
   ```

   Here, we've given the path the name "**oval**" and used an ellipse shape as the path. To apply it to the image we just placed, add an inline style with the **clip-path** property that points to "**oval**":

   ```
   <image x="0" y="0" width="300" height="400" xlink:href="circuits.jpg"
   style="clip-path:url(#oval)"/>
   ```

The teal text isn't legible against the photo, so you can change the text to white or just delete it, as I've done (we all know Bobby's name now). Your SVG should look something like the one in FIGURE 28-8.

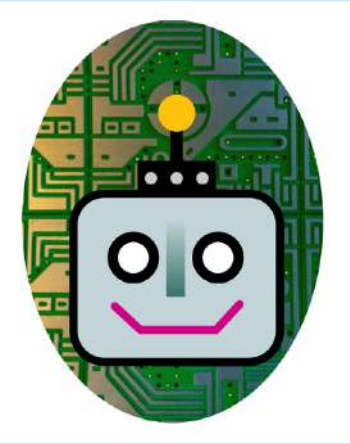


FIGURE 28-8. Bobby the Robot has a spiffy oval background with a circuit board photo.

Filter effects

You might be surprised to learn that a vector-based graphic format includes Photoshop-like filters for manipulating images. The SVG spec includes more than a dozen filter effects that can be used alone or layered and combined for all sorts of effects, such as simple Gaussian blurs, color shifting, mosaic-like patterns, and good ol' drop shadows.

The nice thing about filters is that the original image is untouched; all the messing around happens when the browser renders the image with the filters applied. FIGURE 28-9 shows just a few SVG filters to give you an idea of what can be done. On the downside, filters require a lot of processing power and may slow down performance, especially on mobile devices. Filters and any effects that require generating large raster areas should be used minimally and with care to avoid a performance hit.

To give you a brief taste of how filters work, here is an example that puts a blur effect on an **ellipse** element. The filter is defined with a **filter** element, which contains one or more **filter primitives** (a very specific effect that can be combined with other effects, indicated by the **fe** prefix). The filter is given an **id** and then is called in as a style on the element that uses it:

```
<defs>
  <filter id="blurry">
   <feGaussianBlur in="SourceGraphic" stdDeviation="4"/>
  </filter>
</defs>

 <ellipse cx="200" cy="50" rx="150" ry="100" style="fill: orange;"/>

 <ellipse cx="200" cy="300" rx="150" ry="100" style="fill: orange;
filter: url(#blurry);"/>
```

NOTE

This example and the next one use the **style** *attribute (the same one we use in HTML) to add inline styles to elements.*

Original image

Gaussian blur

Color matrix: saturate

Morphology

Turbulence + Displacement map

FIGURE 28-9. Examples of SVG filters

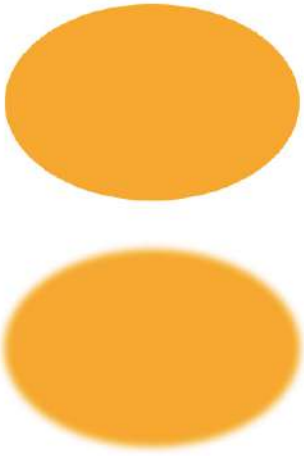


FIGURE 28-10. A Gaussian blur filter applied to an `ellipse` element

FIGURE 28-11. A drop shadow created with SVG filters

■ FUN FACT

The masking, filtering, and transforming features in CSS are extensions of SVG. The standards bodies are aiming to make SVG and CSS work together as seamlessly as possible and to bring some of the best aspects of SVG into CSS and standard browser behavior.

FIGURE 28-10 shows the ellipse without and with the blur filter.

To make a drop shadow, the blur filter is defined and then merged with an offset that moves it down and to the right:

```
<defs>
<filter id="shadow">
  <feGaussianBlur in="SourceAlpha" stdDeviation="4" result="blur"/>
  <feOffset in="blur" dx="7" dy="5" result="offsetBlur" />
  <feMerge>
    <feMergeNode in="offsetBlur"/>
    <feMergeNode in="SourceGraphic"/>
  </feMerge>
</filter>
</defs>
<polygon points="390,12 440,154 590,157 470,250 513,393 390,307
266,393 310,248 189,157 340,154 390,12" style="fill: pink; filter:
url(#shadow)"/>
```

FIGURE 28-11 shows the result.

Of course, there is a *lot* more to SVG filters than I can cover here, but I hope that I've provided a good introduction. Let's pause here for EXERCISE 28-3, where we'll add a blur effect to light up Bobby the Robot's beacon.

Reuse and recycle

A powerful feature in SVG is the ability to define a shape or effect once and then reuse it wherever you need it, as many times as you need it. This keeps the file small by removing redundant code and is a good example of DRY (Don't Repeat Yourself) coding.

The trick is defining the element you want to repeat, such as an icon, in a **`symbol`** element:

```
<symbol id="iconA" viewBox="0 0 44 44">
  <!-- all the paths and shapes that make up the icon -->
</symbol>
```

The symbol does not get rendered; it just sets up a drawing for future use (see **Note**). When you want to use the symbol on the page, call it up with the **`use`** element, which triggers the symbol to render. The following is a minimal **`use`** element example:

```
<use href="#iconA" width="44" height="44" />
```

NOTE

Because symbols won't render, you don't need to put them in a **`defs`** *section, but it's a good practice because it is a logical container for elements you're defining for later use.*

You could include other attributes with instructions such as (x, y) coordinates for positioning and styles that override styles inherited by the copy of the symbol. CSS styles applied to **`#iconA`** will override styles set with attributes in the markup.

EXERCISE 28-3. Adding a glow effect

We're going to make it look like the light on Bobby's antenna is glowing by adding a second, larger circle directly behind the current gold circle and giving it a blur (FIGURE 28-12). As before, we'll define the effect in the **defs** section and apply it to the shape with a style.

1. First, add the following **filter** (named "**glow**") to the **defs** section. Its content is the primitive filter for blurring (**feGaussianBlur**). The standard deviation (**stdDeviation**) value sets the amount of blur. We are using default values, so other filter attributes have been omitted for simplicity:

```
<filter id="glow">
  <feGaussianBlur stdDeviation="10"/>
</filter>
```

2. Next, look for the first **circle** element in the document with the **id** "**beacon**". Copy the element code and paste it just before the existing **circle** element. We'll make a few edits to the values to make this circle a bright blur behind the existing gold circle. Change the **id** value to "**beacon_light**", increase the radius to 36, change the color to a lighter yellow (**#ffc20d**), and use an inline style to apply the "**glow**" filter to the circle:

```
<circle id="beacon_light" cx="150" cy="90" r="36" fill="#f9f208"
  style="filter: url(#glow)"/>
<circle id="beacon" cx="150" cy="90" r="18" fill="#ffc20d"/>
```

You can play around with values for radius, color, and **stdDeviation** for different effects.

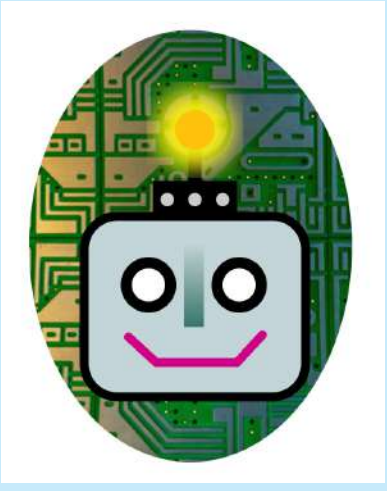


FIGURE 28-12. Adding a glow to the beacon light

The **use** element doesn't work only with **symbol**. You can use it to reuse any basic SVG shape, image, or group in a similar manner. The advantage to making the initial SVG a symbol is that you can include the **viewBox** attribute to enable proportional scaling.

The **symbol** and **use** elements are the tools behind SVG sprites. Sprites are a technique in which multiple smaller images (such as an icon set, to use the most popular example) are placed in a single image file. Code is then used to display just the required portion (for example, a single icon) within the larger image. The technique has its roots in the video game world, but it has become popular for handling image collections on the web as well because it reduces the number of image requests to the server (see the **"CSS Sprites"** sidebar).

In SVG sprites, multiple SVG drawings are defined in one SVG, either in the HTML document or as an external *.svg* file. In the HTML document, the **use** element inside an inline **svg** pulls a particular icon symbol onto the page. This is a powerful tool for managing SVG icons and is often how a design system's icon library is delivered. Hajime Yamasaki Vukelic provides a thorough tutorial on creating and using sprites in his article "Complete Guide to SVG Sprites" (*oreil.ly/D9VHO4*). If you'd like a tool to create them for you, drag and drop your SVGs at *svgsprit.es*.

EXERCISE 28-4 gives you an opportunity to try using symbols within an SVG.

CSS Sprites

We are talking about the SVG sprite technique in this section, but sprites can also be implemented with an inline **img** and CSS. The technique uses a single large background image containing the image collection and the **background-position** property for moving that image around to reveal the part you need. CSS sprites are less popular than they once were because the types of images you use them for (generally icons and UI elements) can be drawn with SVG and handled with SVG sprites. For a more thorough introduction to CSS sprites, see "CSS Sprites: What They Are, Why They're Cool, and How to Use Them" by Chris Coyier (*css-tricks.com/css-sprites/*).

EXERCISE 28-4. Using symbols in an SVG

Because the circles that make up Bobby's eyes are identical, we could define them once as a symbol, then reuse them as needed. Granted, it's only twice in this particular illustration, but that's enough to give you a good idea of the power of symbols.

1. Again, we'll define the symbol in the **defs** section. Add the following **symbol** element:

```
<symbol id="eye" width="50" height="50">
  <circle cx="25" cy="25" r="20" fill="#fff" stroke="#000" stroke-
width="10"/>
</symbol>
```

   Note that the symbol creates a 50-unit square and that the center of the circle that makes up the eye is now positioned in the center of that square.

2. Now we can use the "**eye**" symbol in the drawing. Delete the two **circle** elements that you added earlier for the eyes, and replace them with two **use** elements:

```
<use href="#eye" x="83" y="205"/>
<use href="#eye" x="165" y="205"/>
```

   The symbol is called in by its **id** with the **href** attribute. (If the symbol had been defined in an external SVG document, as is the case for SVG sprites, you would need to use the **xlink:href** attribute, but our symbol is local, so **href** is fine.) The *x* and *y* coordinates position the top-left corner of the square symbol, so you may notice that they are not the same coordinates we used when positioning the **circle** elements directly.

   Save the document and open it in your browser. If all went well, it should look exactly the same as before. "So what's the point?" you may be asking. Let's put the symbol to good use and you'll see.

3. Let's give Bobby blue eyes. Now that we have a symbol for the eyes, we can make a change to the symbol once, and it will repeat everywhere the symbol is used (FIGURE 28-13). Add another **circle** element to the **symbol** to make Bobby's eyes "**steelblue**" or the color of your choice. Save and reload the image, and *presto*! Both eyes are blue!

```
<symbol id="eye" width="50" height="50">
    <circle cx="25" cy="25" r="20" fill="#fff" stroke="#000"
stroke-width="10"/>
    <circle cx="25" cy="25" r="8" fill="steelblue"/>
</symbol>
```

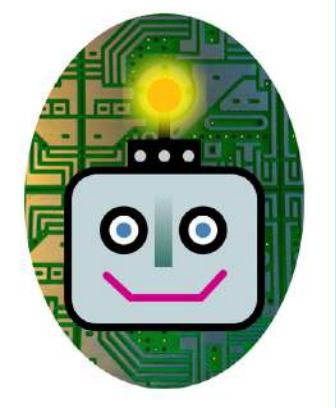


FIGURE 28-13. Adding the blue color to a symbol makes it appear everywhere that symbol is used (in this case, in both eyes).

So far, you've seen SVG used to draw basic shapes, embed images, clip and mask selected areas, and add some pretty groovy special effects. The SVG drawing features are core to the SVG format; however, if we focused only on what gets drawn in the browser window, we would be missing out on some of the best features of SVG. Let's look at them now.

FEATURES OF SVG AS XML

Now you know that behind every SVG that renders on a screen, there is a structured text document. In that respect, it's pretty much the same as HTML. Furthermore, SVG, as a structured document language, has a DOM that includes objects, properties, and methods related to manipulating those elements. This opens up some really exciting possibilities that make SVG more flexible and useful than its static image counterparts.

Styling SVGs

You can target elements in an SVG (or the **svg** element itself, if it's inline) to change their presentation with CSS. Styles are added to SVGs in four ways:

Presentational attributes

The simple SVG example in FIGURE 28-3 used presentational attributes defined in the SVG language, such as **fill** and **stroke-width**, to control how shapes should appear. Presentational attributes are always overridden by styles applied with CSS rules. Here's the example from earlier:

```
<rect x="25" y="25" width="100" height="100" fill="#c6de89"
stroke-width="2" stroke="green"/>
```

Adding SVGs to a Page: A Refresher

Chapter 7, Adding Images, went into detail about the ways to add an SVG to a page, but I thought it would be good to have a brief refresher now in the context of our SVG deep dive. SVGs can be added to an HTML document in the following ways.

As an image

You can add a self-contained *.svg* file to a page with the **img** element, as you would any other graphic:

```
<img src="star.svg" alt="star icon">
```

An SVG can also be used in any CSS property that accepts images, such as **background-image**.

When SVGs are added to pages as simple images, they behave as simple images. You can no longer style or script the elements in the SVG, it won't be interactive (i.e., it won't register user events like clicks or hovering), and it won't load any external files, like embedded images, style sheets, or scripts. If a static image is all you need, this is a reliable option.

As an embedded object

You can use the **object** element to embed an SVG on a page. This method's advantage is that it allows scripts to run and external files to load. You can also provide a fallback image for non-supporting browsers (although there aren't many):

```
<object data="star.svg" type="image/svg+xml">
  <img src="fallback_star.png" alt="">
</object>
```

As inline SVG code

The entire SVG file can be pasted right into the HTML source as an **svg** element. This option gives you total access to the SVG's DOM for styling and scripting, which is a big advantage. On the downside, the code for an SVG can get extremely long, and the image will not be cached for repeated use.

Inline styles

SVG elements may use the inline **style** attribute, which works the same as it does in HTML elements. Many developers prefer this approach. The same **rect** element could also be written as follows:

```
<rect x="25" y="25" width="100" height="100" style="fill:#c6de89;
stroke-width:2; stroke:green;" />
```

Internal style sheet

As in HTML, you can include a **style** element at the top of the **svg** (or in the **defs** section if there is one) that contains all the styles used in the SVG document:

```
<svg> <!-- XML business omitted for brevity -->
  <style>
    rect {
      fill: #c6de89;
      stroke-width: 2;
      stroke: green;
    }
  </style>
  <!-- drawing here -->
</svg>
```

External style sheet

If your SVG is inline or placed on the page with the **object** or **iframe** elements, you can import an external style sheet with the **@import** rule in the **style** element:

```
<svg>
  <style type="text/css">
    @import "svg-style.css";
    /* more styles */
  </style>
  <!-- drawing here -->
</svg>
```

Remember that external files won't work for SVGs embedded with the **img** element. The restriction includes the style sheet itself as well as references to external resources using the **url()** notation within style rules. For inline SVGs, you can also style elements with a style sheet linked to the HTML document with the **link** element:

```
<head>
  <!-- additional head elements -->
  <link href="svg-style.css" rel="stylesheet" type="text/css">
</head>
<body>
  <svg>
  <!-- drawing here -->
  </svg>
</body>
```

Being able to style elements within an SVG along with other elements in the document opens up opportunities to make styles consistent (for example,

applying the same color to HTML elements on the page as well as shapes in the SVG) or even change their appearance dynamically (such as switching to styles designed for "dark mode"). For a comprehensive list of CSS and SVG properties, see "SVG Properties in CSS" by Katherine Kato (*css-tricks.com/svg-properties-and-css/*).

Interactivity with JavaScript

SVGs aren't just pretty pictures—SVGs are images you can *program*! You can add interactivity to the elements in an SVG with JavaScript because all of its element and attribute nodes are accessible in the DOM. It is worth noting that SVGs may also include simple **a** links, which are a basic kind of interactivity.

For example, because an SVG can listen for mouseover (hover) events, you can create fun hover effects that add personality to UI elements. You can also trigger changes in the SVG on a click or a tap. JavaScript can do everything from adding a little motion to an icon to creating complex game interfaces and multimedia presentations, as shown in FIGURE 28-14.

FIGURE 28-14. An example of an SVG game interface created with the Snap.svg JavaScript library. When you put the pointer over each dot, a funny little worm pops up. See more interactive SVG demos at *snapsvg.io/demos*.

If your SVG is inline (added to the document as an **svg** element), scripts in the HTML document can access elements within the SVG. For an SVG embedded in the document with the **object** element, you can use the **script**

TIP

To make an SVG (or part of it) match the color of the surrounding text, use **currentColor** as the value for fill:

```
svg {
  fill: currentColor;
}
```

For more information on this technique, see "Cascading SVG Fill Color" by Chris Coyier (*css-tricks.com/cascading-svg-fill-color/*).

element within the SVG. Because it is an XML document, the script code needs to be wrapped in an **XML Character Data block** (`<![CDATA[ ]]>`) so `<`, `>`, and `&` symbols are parsed correctly as JavaScript syntax (not reserved markup characters), as shown in this example:

```
<script><![CDATA[

  //script here

]]></script>
```

Animation

SVG is a popular choice for adding animated elements to a web page. FIGURE 28-15 is my attempt to capture some charming animation still images. For more inspiration, I recommend going to ***codepen.io*** and searching for "SVG animation." I also recommend the book *SVG Animations* by Sara Drasner (O'Reilly) if you want more how-to information.

Chris Gannon (*codepen.io/chrisgannon/pen/emVgMg*)

Kyle Stark (*codepen.io/kaisle/pen/ZEGwpLb*)

FIGURE 28-15. Examples of animated SVGs. I encourage you to go see them in action at ***codepen.io***.

SVG Versus Canvas

In **Chapter 10, Embedded Media**, we looked at the **canvas** element and the associated API that creates a space for a two-dimensional, dynamic drawing on a web page. The difference is that an SVG image is drawn with a structural markup language and a canvas is drawn with JavaScript commands. Both can contain images, videos, and animation and support dynamic updates in real time.

canvas is better for quick redraws on the fly (it's only pixels, after all), making it better suited for designing games, editing images, and saving images to bitmapped formats. SVG offers advantages in the ease of scripting, animation, and accessibility; however, complicated SVG documents require more processing power than **canvas** elements.

There are two and a half methods for animating SVGs (the third one is only a half option because it's possible, but not recommended going forward):

CSS animation

SVG elements can be animated with the CSS transitions and `@keyframes` rules that you learned about in **Chapter 20, Transitions, Transforms, and Animation**. CSS animation is best for simple animations with three sequences or fewer. It should also be noted that CSS can animate only CSS properties, not attribute values in the markup, which may be limiting. Browser support for SVG animation with CSS is solid, with the exception of Internet Explorer and browsers released before 2017.

JavaScript

JavaScript offers the most options for accessing and manipulating SVG elements and attributes, enabling complex, interactive animations. You can also restyle elements dynamically, based on data input, user interactions, or changes in the environment. Browser support for JavaScript-driven animations is reliable overall, although you may find inconsistent support for animating filters, gradients, and morphing. If you don't want to reinvent the wheel, you can take advantage of an SVG animation JavaScript library (see the **"SVG Animation Tools"** sidebar).

SVG Animation Tools

These are just a few of the JavaScript libraries available to help you add animation effects to your SVGs more efficiently:

GSAP (GreenSock Animation Platform; gsap.com)

[**Animation method:** JavaScript] GSAP is the industry standard for sophisticated SVG animation. You can try it out for free, but to get access to more plug-ins, you'll have to pay a license fee. You'll need some JavaScript and build tool experience to get going with GSAP, but excellent tutorials are available on the website.

SVGator (svgator.com)

[**Animation method:** CSS or JavaScript] SVGator is an online tool for drawing and animating SVGs. It writes all the code for you using your choice of CSS or JavaScript, which makes it a nice tool for beginners to try their hand at animated SVGs.

SVG Artista (svgartista.net)

[**Animation method:** CSS] If you just need a simple, noninteractive animation effect, try uploading your SVG to this online CSS animation tool.

Path animators

Have you ever seen a graphic on a web page that appears to get drawn before your eyes? It was likely an SVG animated with a path animator. There are a number of one-trick-pony JavaScript libraries that can animate the lines in your SVGs, including Walkway (*connoratherton.com/walkway*) and Vivus (*maxwellito.github.io/vivus*).

EXERCISE 28-5. Adding a little animation

The glow around the antenna beacon on our robot is a nice touch, but it would be really cool if it pulsated on and off. We can use CSS keyframe animation to do exactly that. Look back at **Chapter 20** if you need a refresher.

Start by adding a style sheet in the `defs` section with the `style` element. The style sheet contains the `@keyframes` rule for changing the opacity from 0% to 100%. Then, we apply that keyframe animation to the `#beacon_light` circle (the one with the blur) and specify the other animation properties. Here are the styles for the simple fade-in/fade-out animation:

```
<style>
  @keyframes pulse {
    from { opacity: 0% }
    to { opacity: 100%; }
  }

  #beacon_light {
    animation: pulse 1.5s
linear infinite alternate;
  }
</style>
```

Now when you reload the SVG, the glow will fade "on" and "off." For readers who haven't worked along with this exercise, you can use your imagination or open the finished file, *robot_exercise5.svg*, in your browser and see for yourself.

SVG/SMIL

The SVG specification includes animation effects based on **SMIL** (**Synchronized Multimedia Integration Language**), an XML language for creating synchronized audio, video, and animated elements. For example, you can include the `animate` element in the SVG code to animate the value of an attribute over time, or `set` to set its value after a certain time interval.

Although the built-in SVG/SMIL animation elements provide good tools for all sorts of animation tasks, SMIL-based animation is falling out of favor with developers and may not be as well supported by browsers in the future.

Before moving on, perhaps you'd like to try your hand at a simple SVG animation in EXERCISE 28-5.

Data Visualization

SVGs have become a go-to tool for data visualization ("dataviz") because they can be generated dynamically with real data. For example, you could make the temperature level on an SVG thermometer illustration rise and fall with real weather data gathered in the user's location, or you could change progress bars or pie charts as data updates in real time. FIGURE 28-16 shows examples of SVGs used for dataviz from the D3.js gallery. D3.js is a JavaScript library created specifically for "data-driven documents." Find out more at *d3js.org*.

FIGURE 28-16. Examples of data-generated SVGs on the D3js.org site. See more at *observablehq.com/@d3/gallery*.

The methods for generating SVGs with data are dependent on the type of data and the programming language used. One option is to translate the XML document that contains the data into the SVG by using XSLT. **XSLT** (**eXtensible Stylesheet Language Transformations**) is an XML language that provides structured instructions for translating one XML language into another. Clearly, that is some advanced XML mojo that we won't be delving into here, but it doesn't hurt to know. Other options include JavaScript and server-side template languages (also more than we'll take on here).

If you are eager to know more about data-generated SVGs, the book *Interactive Data Visualization for the Web*, 2nd edition, by Scott Murray (O'Reilly) is a good place to start.

Accessibility

Unlike text in bitmap image formats, the text in SVGs can be accessed by search engines and read by screen readers when labeled properly. There are a few measures you can take to make your SVGs more accessible:

- Make sure the SVG meets color contrast guidelines. You can plug foreground and background color values into an online contrast checker, such as the ones provided by WebAIM (*webaim.org/resources/contrastchecker*) or Polypane (*polypane.app/color-contrast*), to be sure it passes the test.
- If the SVG is placed with the `img` element, include `role="img"` and the `alt` attribute with appropriate alternative text in the `img` tag.
- Use the SVG `title` element to provide a short name for the `svg` element itself or any container (such as `g`) or element it contains. The `title` element should be the first child element of its parent.
- Use the SVG `desc` element to provide long text descriptions for elements if beneficial; for example, to describe the content of a chart.
- Add ARIA roles to the `svg` element and its components to ensure that screen readers interpret them correctly and efficiently. Recommendations include the following:
  - An SVG has an implicit ARIA role of `graphics-document`. If it is being used solely as a single, noninteractive image, you can set `role="img"` on the `svg` element.
  - Add `aria-labelledby="title-id"` to the `svg` element to explicitly identify the `title` element as the label of the SVG. The related `aria-describedby` attribute is appropriate for longer descriptions.
  - If elements within the SVG should be accessible (text, links, interactive elements, etc.), do not set a role on `svg`, but add `role="presentation"` to shapes (such as `circle`) and paths to prevent screen readers from announcing the occurrence of every shape in the graphic.

The following articles present examples of common SVG use patterns and how they should be marked up to improve accessibility:

- "Accessible SVGs" by Heather Migliorisi (*css-tricks.com/accessible-svgs/*)
- "Accessible SVGs: Perfect Patterns for Screen Reader Users" by Carie Fisher (*oreil.ly/XVIJ9*)

SVG DESIGN TOOLS

Technically, all you need to create SVG graphics is a text editor (and genius visualization skills plus heroic patience!), but you'll be much happier having a graphics program do the heavy lifting for you. It is also common to create an illustration in a vector illustration program and then bring it into a text editor to clean up the code and add scripts and styles manually. It's a matter of preference, based on your skills and interests.

The most appropriate tool for creating SVGs is a vector-based tool such as Adobe Illustrator or Figma, although even bitmap image editors like Photoshop and GIMP use shapes that can be exported as SVG. These days, there are many vector software options, ranging from free to pricey, and bare-bones to full-featured. Here is a brief roundup.

Vector Illustration Tools

> **TIP**
>
> If you copy a shape in Illustrator, you can paste it as code into a text editor. This is a sweet shortcut if you are doing most of your SVG work in code.

Because SVG is a vector format, it makes sense that vector-based illustration tools such as Adobe Illustrator, Affinity Designer, and Corel Draw are appropriate environments for creating and exporting SVGs. However, these tools often add extra, unnecessary code (even when exporting an optimized version), so be sure to do a search for how to create optimized SVGs in the tool of your choice. Folks have published all sorts of tips that will help your design and production process. You can also optimize the resulting SVG after it's exported, as I'll discuss soon.

Vector-Based UI Design Tools

Many designers choose to design sites and apps in an interface design tool such as Figma or Sketch. Because they are vector-based, it is easy to export page elements in SVG format, but again, it is wise to run the resulting SVGs through an optimization tool to remove unnecessary code. In Sketch, if you control-click on a shape layer, you are given an option to copy the SVG code for that element. If you use Figma, you'll find many SVG-related plug-ins, including several for creating SVG animations.

SVG-Specific Tools

There are a number of nifty little SVG-specific drawing tools available for little or no investment. Because they are designed just for SVG, they have a manageable number of features and settings that map to SVG capabilities (no wading through hundreds of features you don't need). Some of them allow you to view and edit the underlying SVG code as well. Here are a couple of SVG editors of note:

Inkscape (inkscape.org)

Inkscape (FIGURE 28-17) is an open source image editor made specifically for SVG (SVG is its native format). It is available for Windows, macOS, and Linux. Inkscape's interface has evolved quite a bit over the last few years, but if you are accustomed to Adobe tools, it may take a little getting used to. You definitely can't beat the price (free)!

FIGURE 28-17. Inkscape was created for SVGs, so its interface features are SVG-centric, including menu items for Fill and Stroke, Symbols, Clip, and Mask.

Boxy (boxy-svg.com)

Boxy (FIGURE 28-18, left) is a full-featured graphics program available for macOS, Windows, and Linux for a modest $10/year (as of this writing). It has an easy-to-use interface as well as a code inspector.

SVGEdit (github.com/SVG-Edit/svgedit)

SVG-Edit (FIGURE 28-18, right) works right in the browser. It provides all the basic drawing tools, layers, and the ability to view and edit the SVG source. You can also export images to PNG, JPEG, BMP, and WebP. It's free, so there's no reason not to give it a try; just click one of the links under "I want to use SVGEdit" on the tool's GitHub page.

TIP

When saving an SVG in Inkscape, choose Plain or Optimized. The Inkscape SVG option contains a lot of proprietary code that unnecessarily drives up the file size. Be aware that the Optimized version may strip out more than you want it to, depending on what you intend to do with the graphic.

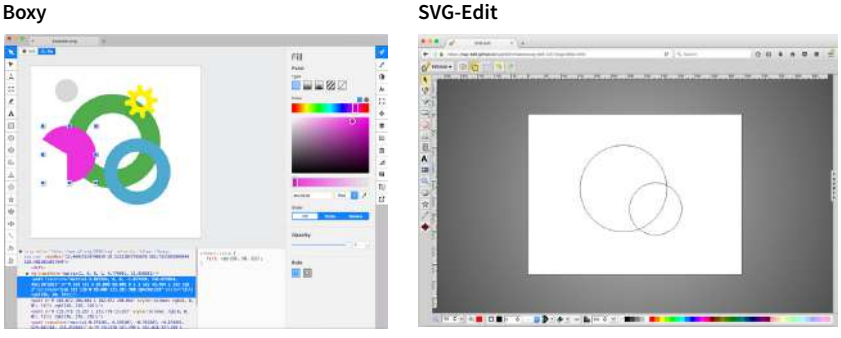


FIGURE 28-18. SVG creation tools Boxy and SVG-Edit

Code Editors

If writing SVG by hand is your thing, or if you need to tinker with preexisting SVGs, you may be able to see your SVG render as you type using a plug-in right in your favorite coding tool. I have confirmed that SVG preview plug-ins are available for Visual Studio and Sublime Text, but you can do a search to see if there is a similar option for your favorite editor.

SVG PRODUCTION TIPS

When you create an SVG drawing, keep in mind that the end result is a text file. If you have plans to style the SVG with CSS or manipulate it with JavaScript, you want the markup to be as well structured as possible. As with any file that is being delivered on a web page, you also want that file to be as small as possible.

The downside to using standard vector illustration tools is that you don't have much control over the code they output. Most of them export SVG code that is inefficient, redundant, and full of proprietary cruft. Fortunately, there are measures you can take while designing and after you export the SVG to ensure it is as lean and mean as possible. Many a blog post has been written about SVG optimization, but the following tips, although not comprehensive, should point you in the right direction. Ultimately, you'll need to get familiar with your chosen tool's quirks to anticipate and correct its shortcomings.

Pre-Export Best Practices

Decisions you make in the design space can improve your SVG output in terms of organization and file size. Here are some production tips that will help you create SVGs with the best quality at the smallest file sizes:

Define the artboard or drawing size in pixels.

The artboard dimensions correspond to the viewport (**`width`** and **`height`** attributes) of the **`svg`** element.

Don't make the artboard larger than it needs to be.

The artboard will be used as the viewport that displays the SVG content, so don't make it larger than it needs to be. The caveat is that you should leave a few pixels of space around the drawing to allow for anti-aliased edges that may appear when the SVG is scaled. If the path abuts the canvas edge precisely, the soft anti-aliased pixels may be cut off.

Use layers to group elements logically.

If you plan on animating or scripting your SVG, keeping your design document well organized as you work will help with accessing the pieces you need later. In Illustrator and most other tools, layers are converted to group (**`g`**) elements, and nested layers are stored as nested groups.

Give elements and layers meaningful names.

The names you give elements and layers are used as **`class`** and **`id`** values in the SVG code, so make sure they are descriptive. Names should be all lowercase with no character spaces so they are appropriate as attribute values.

Simplify paths.

The more points and handles are used to define a path, the more coordinates appear in the SVG source. More coordinates means more characters and a larger file size. Take advantage of any "simplify path" function your tool offers. Also consider using methods that reduce the number of elements in the file, such as merging objects that always appear as one unit and using one wide stroke instead of two strokes and a fill. If possible, ensure your tool uses shape elements like **`circle`** and **`rect`** instead of multipoint paths for simple shapes. If your tool does not provide a way to do this, it is possible to replace the path code with the simple shape element manually.

Be aware of decimal places.

Keeping in mind that more characters result in a larger file size, you can shave bytes off your file by limiting the number of characters after the decimal points in your designs. For example, storing "100.3004598, 248.9998765" as coordinates requires more data than simply "100.3, 249". Many tools allow you to limit the number of decimal places on export. You may also choose to "snap to pixels" when you set up the document so that points always fall on even integers. The general rule of thumb is that the smaller the image is, the more decimal places are required to accurately define the points in objects. Large images can tolerate lower accuracy without compromising quality. You may need to experiment to find the right balance of decimal places and quality for your image, but two decimal places is a good starting point.

Avoid raster effects inside the SVG.

SVG's efficiency lies in the fact that it is vector-based. When you introduce raster (bitmap) image material to the SVG, that file size advantage is lost. Be aware that certain effects in drawing programs—such as blurs, drop shadows, glows, and so on—often generate a raster image area when you may not be expecting it, adding greatly to the file size. If you find that your *.svg* file is unusually large, unexpected raster image material is a likely suspect.

Pay attention to fonts.

As with any font style suggestion on the web, there is no guarantee that your chosen font will be available on the user's machine, so the font you use in your SVG may not display. Be sure to test and provide fallbacks. If there is a small amount of text, and it doesn't need to be searchable or read aloud by an assistive device, consider whether converting the text to paths is a better option.

Use centered strokes.

Although not related to file size, you will achieve better results if you design with strokes set to be centered over the path because that is how SVG handles strokes natively. Some tools make adjustments to compensate for an outside or inside stroke, but just centering your strokes is a better starting point.

Of course, there are many tool-specific tips that I am not able to cover here, so do a search for SVG optimization tips for the tool of your choice.

Post-Export Optimization

Even with careful planning up front, exported SVGs tend to have a lot of redundant code, unnecessary metadata, hidden elements, and other fluff that can be safely removed without affecting the way they render. It's a good idea to run your SVGs through an optimizer to strip all that out and slim down the file size. The following are just a few of the available options.

> **FURTHER READING**
>
> Chris Coyier's article "Tools for Optimizing SVG" on CSS-Tricks provides a nice little roundup of optimization options for SVGs (*css-tricks.com/tools-for-optimizing-svg/*).

SVGO

The biggest SVG optimization player in town as of this writing is SVGO (*github.com/svg/svgo*). It can make individual tweaks—such as removing empty attributes, removing the `xmlns` attribute (just fine if the SVG is used inline), removing comments, and dozens more—and you can pick and choose how you want to squeeze down the file, depending on its end use.

The best thing about SVGO is that there are so many ways to use it! There are SVGO plug-ins for Illustrator (SVGNow), Inkscape (SVGO-Inkscape), Figma (Advanced SVG Export), and Sketch (SVGO Compressor). You can use it via the command line or within a build tool or task runner (like Grunt), and

it's also available as a macOS folder action, where optimization is performed when you drag files into it. The full list of options is listed on the SVGO site.

One of the easiest ways to see SVGO in action is to use the web-based SVGOMG tool created by Jake Archibald, which provides a graphical user interface for SVGO (*jakearchibald.github.io/svgomg*). SVGOMG lets you toggle the various optimization plug-ins individually and view the results in a live preview (FIGURE 28-19), which is a handy feature during development.

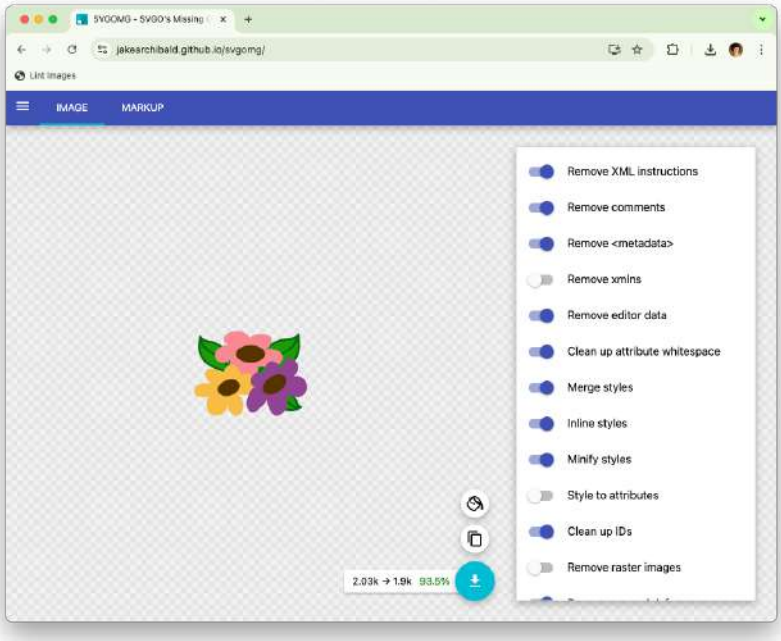


FIGURE 28-19. SVGOMG allows you to select optimization options individually.

File compression tools

Because they are text files, SVGs can be compressed with tools such as Gzip or Brotli that run on the server and compress text files with fancy algorithms. You'll need to configure the server to use them, which is beyond the scope of this chapter, but you can find plenty of tutorials online. The file savings are worth the effort.

Gzip (*gzip.org*) has been around as long as the web itself. Gzipped SVGs (which use the suffix *.svgz*) can be as little as 15–25% of their original size.

Brotli (*github.com/google/brotli*) is an open source compression algorithm released by Google in 2015 that is giving Gzip a run for its money in terms of compression and performance. In fact, as of this writing, Brotli is used more than Gzip for JavaScript and CSS.

SCALING SVGS

I've talked a lot about how the scalable nature of SVGs makes them great for use in responsive layouts, where they can scale to fit changing element widths with no loss of quality. Resizing SVGs—whether to a specific size or automatically to fill the available space in the layout—is not quite as straightforward as resizing bitmap images. We'll begin our resizing discussion with a look at the SVG features related to scaling.

The dimensions of the drawing space and the viewport work independently of one another.

The Viewport, viewBox, and preserveAspectRatio

To understand how SVGs scale, it is useful to have the proper mental model of how they display. There are two components to how an SVG displays in the browser: the drawing itself (also called the **user space**, where the content of the SVG is created), and the **viewport** that appears in the layout and defines the visible areas of the user space.

The viewport window

The viewport acts like a window through which you view the drawing space, similar to a browser viewport or an `iframe` element displaying an HTML document. Just as there is no guarantee that the *entire* HTML document will fit inside the browser window or `iframe`, there is no guarantee that the entire SVG drawing will fit perfectly within its viewport. The drawing might be smaller and leave extra space; it might be bigger and get cropped off. And here's the important part: *the dimensions of the drawing space and the viewport work independently of one another*.

NOTE

In this section, I will be omitting the `version` *and* `xmlns` *attributes in the* `<svg>` *tag as well as the SVG content to keep the examples compact and clear.*

You set the dimensions of the SVG viewport with the `width` and `height` attributes on the `svg` element (see **Note**). You can omit the `px` because SVG user units equal one pixel by default:

```
<svg width="400" height="300">
```

The default value for `width` and `height` is 100%, so if you omit both attributes the SVG will fill the available space in the window or container. This is the key to making an SVG responsive, as you'll see soon. When an SVG without dimension attributes is embedded on the page with `img`, `object`, or `iframe`, the sizing attributes and styles applied to the embedding element establish the viewport size.

Earlier in this chapter (FIGURE 28-4), we saw that the viewport uses a set of coordinates (the **viewport coordinate system**) that start at 0 in the top-left corner and increase to the right and downward in 1-unit/pixel increments. Keep that image in mind when we talk about positioning and scaling.

The viewBox attribute

The drawing (user) space is like an infinite vector space measured in flexible user units. That drawing space has its own set of coordinates (the **user coordinate system**). By default, a user unit is equal to a pixel, but you can change the scale (and thus the scale of the SVG content) with the **viewBox** attribute in the **svg** element. The **viewBox** attribute defines the position, scale, and aspect ratio of the drawing relative to its viewport, so it is super-duper important to our sizing discussion. It is the key to preserving the aspect ratio of the drawing when the SVG resizes in a responsive layout.

The viewBox attribute is the key to preserving the aspect ratio of the drawing.

The syntax for the **viewBox** attribute is as follows:

```
viewBox="x y width height"
```

Values may be separated by space characters, as shown, or by commas.

The *x* and *y* values determine the position of the top-left corner of the user coordinate system within the viewport. Setting them to **0 0** will position the top-left corner of the drawing at the top-left corner of the viewport, which is the default position. You can move the image around with positive or negative number values.

The *width* and *height* values in **viewBox** give the SVG content dimensions and establish its aspect ratio. Unlike bitmap images made up of a finite number of pixels, SVGs don't have an inherent aspect ratio of their own, so we must define it explicitly with **viewBox**. Fortunately, most vector tools that generate SVGs include **viewBox** in the code automatically. If you find your SVG isn't scaling as you expect, you might want to check the **viewBox** settings in the code.

How scaling works

OK, so now you know that the SVG viewport is a little window that we place in the layout through which we can see the SVG drawing content. You also know that the drawing itself has its own coordinate system and we can control its placement and scale with **viewBox**. Let's see how they work together.

The SVG in this example has both its viewport (**width** and **height**) and its drawing space (**viewBox**) set to 400 × 500 pixels:

```
<svg width="400" height="500" viewBox="0 0 400 500">
  <!-- drawing content here -->
</svg>
```

The drawing space matches the viewport (this is the default behavior if **viewBox** is omitted), but let's see what happens when we reduce the size of the drawing space by half (to 200 × 250 pixels):

```
<svg width=400 height=500 viewBox="0 0 200 250">
  <!-- drawing content here -->
</svg>
```

The result is that now 1 user unit in the drawing maps to 2 pixels in the viewport, so the drawing renders twice as large (FIGURE 28-20). The two coordinate systems (viewport and user), which are the same by default, now have different scales. And because the size of the viewport hasn't changed, it crops out the portions of the drawing that don't fit (they still exist; you just can't see them in the "window"). The point of this example is merely to demonstrate that the dimensions and coordinate systems of the viewport and the drawing space work independently of one another and can be used to adjust scale.

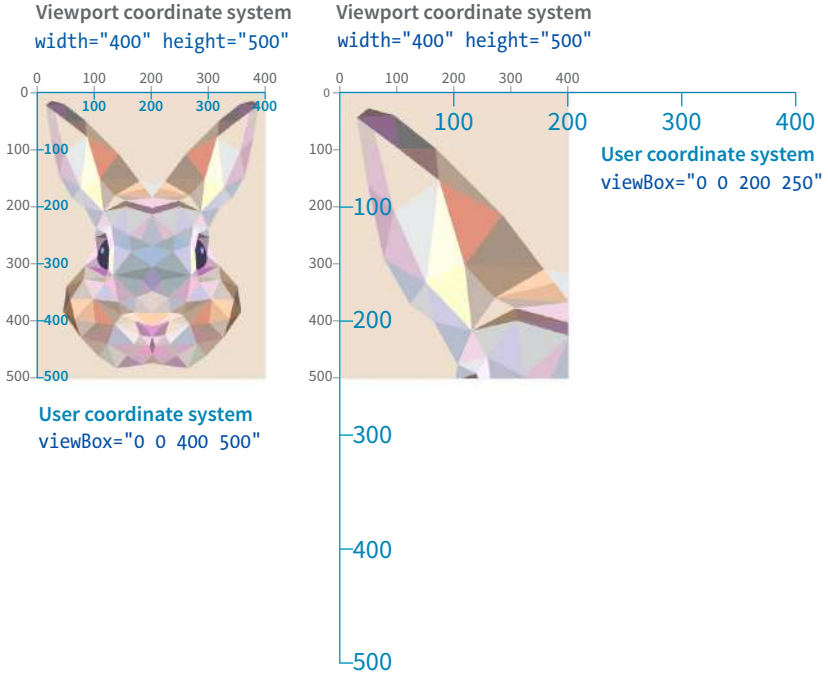


FIGURE 28-20. The viewport (`width` and `height`) and the viewbox (`viewBox`)

I could fill half a chapter describing the full capabilities and effects of the `viewBox` attribute. I want you to know everything about it, but for the sake of space, I will refer you to Sara Soueidan's wonderful tutorial "Understanding SVG Coordinate Systems and Transformations (Part 1)—The viewport, viewBox, and preserveAspectRatio" (*oreil.ly/oVcNL*). You will also find thorough explanations in the SVG books recommended at the end of the chapter.

For the purposes of this responsive SVG discussion, however, just keep in mind that the `viewBox` attribute is the key to establishing the width and height (and resulting aspect ratio) of the SVG drawing itself, independent of the viewport "window" it will be displayed in.

The preserveAspectRatio attribute

The **preserveAspectRatio** attribute in the **svg** element, as the name suggests, is responsible for making sure the drawing maintains its aspect ratio, so it is important to our goal of creating a proportional responsive SVG. **preserveAspectRatio** only works if the **viewBox** attribute is present, as that's what establishes the ratio to preserve.

Conveniently, **preserveAspectRatio** is turned on by default, which means you don't need to include it if your desired effect is to maintain proportions. Setting it **preserveAspectRatio="none"** allows you to stretch and squish the drawing however you like, just as you can a bitmap image.

The **preserveAspectRatio** attribute also controls how the content should be aligned in and fill its viewport. The attribute takes two keyword values separated by a space, as shown in this syntax example:

```
preserveAspectRatio="alignmentKeyword [meet or slice]"
```

By default, content is centered horizontally and vertically in the viewport and sized just big enough to show the entire image, even if some parts of the viewport are left empty. Here's how the default looks written out in code:

```
preserveAspectRatio="xMidYMid meet"
```

The **xMidYMid** keyword places the content at the midpoint of both the x- and y-axes. It is one of a collection of alignment keywords. To position the image all the way to the left and against the bottom axis, you would use the keyword **xMinYMax** (**Min** is equivalent to 0%; **Max** is equivalent to 100%). That should give you a general idea of how the system works, however, for most instances, the default is fine.

The **meet** keyword sizes the content until it meets either the height or the width of the viewport (similar to **contain** for background images). The result is that the entire image will be visible, but there may be space left over in the viewport if the aspect ratios do not match. The alternative is **slice**, which sizes the drawing so that it covers the entire viewport, even if some of the drawing is cut off (like **cover** for background images).

I recommend you get to know the capabilities of **preserveAspectRatio** on your own, but for now, know that it is the attribute that does the work of keeping the aspect ratio intact, even if you take advantage of the default setting and don't include it explicitly.

Now that you are familiar with the SVG scaling mechanisms at work behind the scenes, let's look at techniques for a few common sizing scenarios:

- Sizing the SVG to fixed dimensions
- Making a responsive SVG embedded with **img** or **object**
- Making a responsive inline SVG (**svg**)
- Using an SVG as a responsive CSS background image

Sizing to a Fixed Dimension

Perhaps you have a small image on your page that will always appear at the same dimensions regardless of the screen it's viewed on, such as an icon. This one is easy. If it is an inline SVG (placed with the **svg** element), use the **width** and **height** attributes to specify the viewport size in the layout (see **Note**):

```
<svg width="160" height="120" viewbox="0 0 160 120" ... >
```

For SVGs embedded with the **img** element, set the dimensions of the **img** element (with **width** and **height** attributes or CSS styles), which in turn establishes the viewport size for the embedded SVG (FIGURE 28-21). Note that I've added a dotted outline in the figure to indicate the edges of the viewport, but this would not be visible ordinarily:

```
<img src="flowers.svg" width="160" height="120" alt="">
```

NOTE

CSS **width** *and* **height** *properties applied to the* **svg** *element in a style sheet override* **width** *and* **height** *attribute values.*

The dimensions of the **img** element match the **viewBox** dimensions:

```
<img src="flowers.svg" width="160" height="120" alt="">
```

When the **img** element doesn't match the **viewBox** dimensions, the image scales to meet the nearest edges of the viewport and is centered within it.

```
<img src="flowers.svg" width="300" height="150" alt="">
```

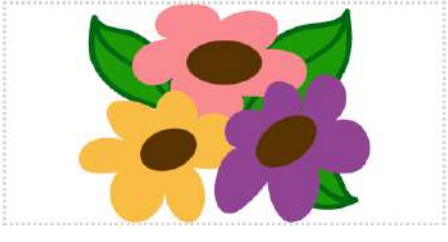


FIGURE 28-21. SVGs placed on the page with the **img** element use the dimensions of the **img** for the viewport.

If you provide only one dimension, modern browsers handle the other as if it is set to **auto**, which means the viewport will auto-size based on the ratio established by **viewBox**. Older browsers and Internet Explorer may not behave that way, so it is safest to provide both dimensions for SVGs you want to display at a specific size.

If you provide image dimensions that *don't* match the aspect ratio defined in the SVG, the default **preserveAspectRatio** setting kicks in and the drawing content will be proportional, centered, and fully visible (**meet**), as shown in FIGURE 28-21 (bottom).

Responsive SVGs

If your layout is responsive, you'll likely want your SVGs to resize proportionally to fill the available space. I'll be referring to this use case as "responsive SVGs," and they are easy to achieve in modern browsers (see the **"Responsive SVG Browser Support"** sidebar).

When it comes to responsive SVGs, the **viewBox** attribute is *everything*, so make sure your SVG has one. Even if the drawing ends up a different width and height in the final layout, **viewBox** establishes the aspect ratio. If you want to keep the image at that aspect ratio, you don't need to do anything because **preserveAspectRatio** takes care of it by default. The other key is omitting the **width** and **height** attributes in the **svg** element so it will use its default 100% values.

In this section, we'll look at how to make SVGs responsive when they are embedded with **img** or **object**, placed inline with the **svg** element, and used as a CSS background image.

Responsive SVG Browser Support

Older browsers and all versions of Internet Explorer have hiccups when it comes to auto-sizing SVGs, so there are a few extra hoops to jump through if you have a reason to support them. For full details I will refer you to Amelia Bellamy-Royd's article "How to Scale SVG" on CSS-Tricks (*css-tricks.com/scale-svg/*), which includes some workarounds to get SVGs to resize predictably in all browsers. I've made note of a few tweaks here, but I've omitted complicated hacks.

Responsive SVGs embedded with img or object

By default, the width and height of an SVG file is 100%, so we can take advantage of that behavior to force the **img** to fill the width of its containing element (see **Note**). As mentioned earlier, the important steps are:

- Do *not* include **width** and **height** attributes in the SVG source code, so the default 100% value will be used.
- Make sure the SVG includes a **viewBox** attribute to establish the aspect ratio (it will be preserved by default).

In the following example, the **img** element is in a **div** set to 50% of the browser window's width, as might occur in a responsive layout (FIGURE 28-22). Note that there are no **width** and **height** attributes in the SVG source, but the **viewBox** attribute is included. As the window gets larger and smaller, the SVG scales with it, maintaining that crisp vector quality at all scales. I added a dashed outline on the **div** to reveal its boundaries in the figure.

THE SVG SOURCE *(flowers.svg)*

```
<svg version="1.1" viewBox="0 0 160 120" ...>
  <!-- flower drawing -->
</svg>
```

THE HTML MARKUP

```
<div class="container">
  <img src="flowers.svg" alt="flower illustration">
</div>
```

THE STYLES

```
.container {
  width: 50%;
  outline: 1px dotted gray;
}
```

NOTE

To support older browsers and Internet Explorer, you must explicitly set the width and height of the **img** *element to make the SVG scale proportionally:*

```
img {width: 100%; height: auto;}
```

FIGURE 28-22. The SVG scales automatically and proportionally with the size of its container.

Responsive inline SVGs

If you want an inline SVG to automatically resize to fill the width of its container, the setup is the same:

- Omit the **width** and **height** attributes in the **svg** source code.
- Include the **viewBox** attribute.

To re-create the resizing behavior shown FIGURE 28-22, put the inline SVG directly in a container **div**. Its 100% default width and height ensure it will fill the available space (see **Note**).

THE HTML MARKUP

```
<div class="container">
   <svg version="1.1" viewBox="0 0 160 120">
     <!-- flower drawing -->
   </svg>
</div>
```

THE STYLES

```
.container {
  width: 50%;
  outline: 1px dotted gray;
}
```

NOTE

To support older browsers and Internet Explorer, you must employ a padding hack that holds the **div** *open at the proper aspect ratio, similar to the one covered in the next section. The hack, should you need it, is described in full in the CSS-Tricks article by Amelia Bellamy-Royds, referenced earlier.*

Responsive SVGs as CSS background images

You can use an SVG as a background image, just as you can with any other image format. However, you need to use a "**padding-bottom** hack" to expand the containing element to the aspect ratio of the SVG.

The foundation of this hack is the defined behavior of padding. Padding, when specified in percentage values, is always based on the *width* of the element (even top and bottom values). That means that as the width of the container **div** changes, so too does the amount of padding, always maintaining the same relationship. That sets up a constant aspect ratio for the SVG to fill.

The following example uses bottom padding to create a **div** with a width of 100% and a height calculated from the aspect ratio of the background SVG (160 × 120 pixels). As in the previous responsive examples, the **width** and **height** attributes have been omitted in the SVG source (not shown) and the **viewBox** is included.

THE MARKUP

```
<div class="container">
  <!-- optional content -->
</div>
```

THE STYLES

```
.container {
  width: 100%;
  background-image: url(flowers.svg);
  background-size: cover;
  height: 0;
  padding: 0; /* reset */
  box-sizing: content-box; /* optional */
  padding-bottom: 75%   /* [120/160 x 100%] */
}
```

As you can see, the **width** of the **div** is set to 100% and the background image is set to cover the entire background. Several of the properties here are working as resets to make sure that the padding hack works as intended; for example, the initial **height** is set to 0, and the **padding** is set to 0 to override any possible inherited values. The hack requires **box-sizing** to be set to **content-box**, which is the default, so include this property only if you've set all elements in the document to use **border-box** in an earlier rule.

Now let's look at the **padding-bottom** property for which the hack was named (see **Note**). To calculate the desired height of the **div**, divide the SVG's height by its width and multiply the result by the percentage width of the **div**:

```
(svg-height / svg-width) * div-width%
```

In our example, the padding is set to 75%, which is the result of the formula shown above for our particular SVG [(120 / 160) × 100% = 75%]. You could also let the browser do the calculation for you, although some older browsers don't support it:

```
padding-bottom: calc(100% * 120 / 160);
```

Our example uses a width of 100%, but you can set the width of the containing element to any percentage value—just be sure to use that percentage value in the bottom padding calculation as well. For example, if we wanted the **div** to fill only 50% of the available width of the viewport, the **padding-bottom** value would be 37.5% [(120 / 160) × 50% = 37.5%].

You're nearly to the end of the chapter (and the whole book!), but if you'd like some hands-on practice with SVG scaling, you can work along with the steps in EXERCISE 28-6.

NOTE

You'll sometimes see this hack using **padding-top**, *but the principle is the same. Bottom padding was less problematic for some older browsers, so it is a little more reliable.*

EXERCISE 28-6. Scaling SVGs

There's no better way to wrap your head around how SVGs scale than to try it yourself. Feel free to plug in your own values as well, to see how the image responds.

1. Start by opening the HTML file *scaling.html* (included with the materials for this chapter) in your text or code editor. You'll find that it is a very minimal document containing only an inline SVG of the robot as we left it in the previous exercise. To make it easier to visualize the edges of the viewport, I've added an inline **style** element that applies a dashed outline and a 1em margin on all sides. Here is the opening **<svg>** tag that we're starting with (I've stacked the attributes to make them easier to read and edit):

   ```
   <svg xmlns="http://www.w3.org/2000/svg"
        xmlns:xlink="http://www.w3.org/1999/xlink"
        viewBox="0 0 300 400"
        style="outline: 2px dashed lightblue; margin: 1em"
        width="300" height="400">
   ```

   The **width** and **height** attributes set the dimensions of the viewport, in this case at the same size as the drawing itself (300 × 400 pixels). Before we start editing values, open *scaling.html* in a browser to see the SVG at its original size.

2. Change the **width** value to 100, save, and refresh the browser. You can see the **viewBox** attribute doing its job keeping the image proportional, and the default **preserveAspectRatio** is sizing the image to "meet" the sides of the viewport while centering the drawing in the available space (FIGURE 28-23). Try changing the **width** to 500 and look at it again. You should see that the same things are happening, only now there is extra space on the sides of the image.

3. Now remove the **height** attribute altogether, then save and reload the document in the browser. If you are using a modern browser, you should see that the whole viewport has scaled up proportionally with a width of 500 pixels. That's because if one of the dimension attributes is missing, it is assumed to be **auto**, so the browser automatically sizes the image to the 3:4 aspect ratio established by **viewBox**.

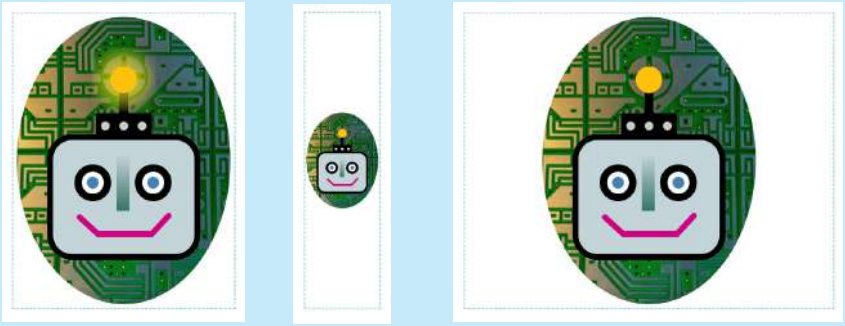


FIGURE 28-23. Playing with **width** and **height** attributes.

EXERCISE 28-6. Continued

4. That was fun. What if we delete the **width** attribute too? The SVG now fills the width of the browser window and scales proportionally. We're seeing the default values of 100% at work here.
5. Let's try something a little bit different. Add the **width** and **height** attributes back in there, set to 100 and 400, respectively (the same values we used in step 2). Also add **preserveAspectRatio="none"** to set Bobby free from his 3:4 aspect ratio. Save and reload the document in your browser. Now the image is squished into the viewport space (FIGURE 28-24, left). To squish him the other way, try setting the **width** to 500 (FIGURE 28-24, right).
6. Finally, we'll make the SVG responsive, resizing to fill a **div** set to 60% of the browser width. The first thing to do is delete the **width** and **height** attributes so the SVG will use the default 100% values and fill the available space in the **div**. Also delete the **preserveAspectRatio** attribute we added in the last step. Next, wrap the inline **svg** in a **div** element with the **id** "**container**":

```
<div id="container">
   <svg ... >
   </svg>
</div>
```

7. In the **style** element in the **head** of the document, set its width to 60vw:

```
<style>
  #container { width: 60vw; }
</style>
```

   Open the file in your browser and try resizing the window. You should find that the SVG resizes proportionally as the width of the window and **div** changes.

That's a wrap on the guided steps for EXERCISE 28-6, but I encourage you to try similar steps with the *robot.svg* file embedded on the page with **img** or **object**.

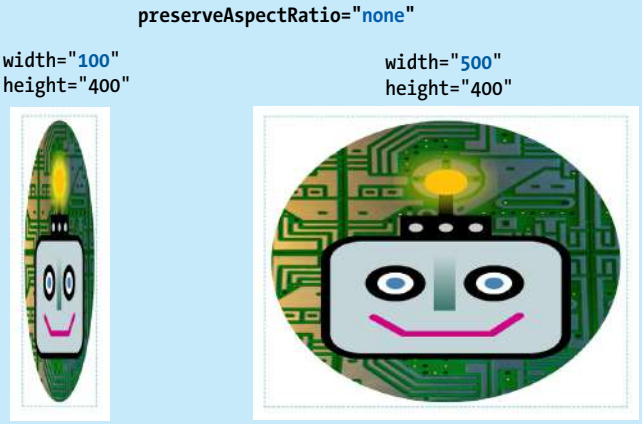


FIGURE 28-24. When **preserveAspectRatio** is set to **none**, we can change the proportions of the drawing in the viewport.

More Fun with Scaling

The examples in this section show the most straightforward type of scaling, in which the entire image resizes larger or smaller. However, because you can access individual objects and properties within the SVG document, you can do some pretty nifty sizing tricks with particular parts of the content. I don't have enough pages for complete tutorials here, but I wanted you at least to know about some options.

Scaling individual parts of the SVG independently

Take advantage of nested **svg** elements and symbols to scale different parts of the image separately. For example, you could make the background responsive, but keep one element (like a logo) the same size. You'll find examples of this technique at the end of Amelia Bellamy-Royd's article "How to Scale SVG" (*css-tricks.com/scale-svg/*).

Scaling an SVG while preserving the stroke width

By default, strokes in an SVG get thicker when the image gets larger (such as when a user zooms in). If you want to prevent that from happening, you can apply the `vector-effect="non-scaling-stroke"` attribute to the path you want to prevent from scaling. As of this writing, it only works when the image is zoomed or if it is scaled using the `transform` or `scale` properties. In other words, it doesn't kick in for responsive SVGs that are just resizing to fit a layout. That's a bit limiting, but it's nice to know this option is available.

Adapting content at different sizes

The problem with simply scaling SVGs (or any image, really) smaller is that details may get lost at small sizes. One approach to solving that problem is to provide simplified versions of icons or logos when the image is very small. **Adaptive icons** (also referred to as **responsive icons**) use media queries to test the viewport size and deliver the appropriate amount of detail. FIGURE 28-25 shows a logo icon created by Sara Soueidan that becomes simplified at smaller sizes. Read how she did it in the article "Making SVGs Responsive with CSS" (*oreil.ly/mny_V*).

FIGURE 28-25. An example of adaptive icons by Sara Soueidan

FURTHER SVG EXPLORATION

Obviously, I was only able to scratch the surface of Scalable Vector Graphics in this chapter. When you are ready to use SVGs on your site, you'll have more brushing up to do. There are many online tutorials available, but these are the resources that helped me the most in my own exploration. Be aware that the books are nearly a decade old, so although the fundamentals will be solid, the browser support information may be dated:

- *SVG Essentials*, 2nd edition, by J. David Eisenberg and Amelia Bellamy-Royds (O'Reilly)
- *Using SVG with CSS3 and HTML5* by Amelia Bellamy-Royds, Kurt Cagle, and Dudley Storey (O'Reilly)
- *Smashing Magazine* (***smashingmagazine.com***) and *CSS-Tricks* (***css-tricks.com***), both of which have many useful articles explaining SVG fundamentals and advanced techniques

TEST YOURSELF

I just threw a lot of SVG info at you. Try taking this quiz to see what stuck. The answers appear in **Appendix A**.

Refer to this SVG element for questions 1 through 4:

```
<rect x="0" y="0" width="600" height="400" />
```

1. What is **`rect`**?

2. What do **`x`** and **`y`** do?

3. What do **`width`** and **`height`** do?

4. Explain the forward slash (**`/`**) character.

Now back to our regularly scheduled questions:

5. What is the primary difference between SVG clipping and masking?

6. Name three ways to reduce the file size of an SVG.

7. What is the most widely supported version of SVG?
   a. SVGTiny
   b. SVG 1.1
   c. SVG 2
   d. SVG, Electric Boogaloo

8. Name three of the five ways you can style elements in an SVG.

9. How do you define the aspect ratio of an SVG drawing?
   a. The `preserveAspectRatio` attribute
   b. The `width` and `height` attributes
   c. The `viewBox` attribute
   d. None of the above

10. Say you want to be able to stretch and distort your SVG image. What code do you need to add to the SVG's source to allow that to happen?

11. Which of the following can be used to animate elements in an SVG?
    a. Special SVG elements for animation
    b. JavaScript
    c. CSS transitions and keyframe animation
    d. All of the above

AND...WE'RE DONE!

Hey! Here we are at the end of the book! I hope that you've enjoyed this detailed tour through HTML, CSS, JavaScript, and image production. I know the content here is pretty dense, but remember that you don't have to learn it all at once, and you don't need to keep it all in your head. Heck, I *wrote* this book, and I still go back to it to look up attributes, properties, and values. With practice, however, a lot of it will become second nature to you.

I hope that you will find plenty of opportunities to put your new knowledge to use. Best of luck in your web endeavors!

VI
APPENDICES

APPENDIX

A

ANSWERS

Chapter 1: Teams and Tools

1. b, d, a, c
2. The W3C guides the development of web-related technologies.
3. c, d, a, b
4. Frontend development is concerned with aspects of the site that appear in or are related to the browser. Backend development involves the applications and databases required on the server for site functionality. With modern web development technologies, those lines have blurred.
5. Git is the program that teams use for version control.
6. Yes. Providing the code for prototypes and page components is part of the designer's responsibility. Even if they don't write the final production code deployed to the server, they are expected to have a familiarity with the medium they are designing for.

Chapter 2: How the Web Works

1. c; 2. j; 3. h; 4. g; 5. f; 6. i; 7. b; 8. a; 9. d; 10. e

Chapter 3: Some Big Concepts You Need to Know

1. There are a number of unknown factors when you're developing a site. Some that were addressed in this chapter include:
   - The size of the screen or browser window
   - The user's internet connection speed
   - Whether the browser supports specific features
   - Whether the user is using assistive devices such as screen readers or alternative input devices

2. c, d, a b
3. The four general disability categories include:
   — Vision impairment: make sure the content is semantic and in logical order for when it is read by a screen reader.
   — Hearing impairment: provide transcripts for audio and video content.
   — Mobility impairment: use measures that help users without a mouse or keyboard.
   — Cognitive impairment: content should be simple and clearly organized.
4. You would use a waterfall chart to evaluate your site's performance in the optimization process.

Chapter 4: Creating a Simple Page

1. A tag is part of the markup (brackets and element name) used to delimit an element. An element consists of the content and its tags.
2. The recommended markup for a minimal HTML document is as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf8">
    <title>Title</title>
  </head>
  <body>
  </body>
</html>
```

3. a. *Sunflower.html*—Yes.
   b. *index.doc*—No, it must end in *.html*.
   c. *cooking home page.html*—No, there may be no character spaces.
   d. *Song_Lyrics.html*—Yes.
   e. *games/rubix.html*—No, there may be no slashes in the name.
   f. *%whatever.html*—No, there may be no percent symbols or special characters.
4. a. The **src** and **alt** attributes are missing:

```
<img src="birthday.jpg" alt="">
```

   b. The slash in the end tag is missing:

```
<em>Congratulations!</em>
```

   c. There should be no attribute in the end tag:

```
<a href="file.html">linked text</a>
```

   d. The backslash should be a forward slash:

```
<p>This is a new paragraph</p>
```

5. Add **<!--** and **-->** tags:

```
<!-- product list begins here -->
```

Chapter 5: Marking Up Text

1. Here is the markup for a thematic break between these paragraphs:

```
<p>People who know me know that I love to cook.</p>
<hr>
<p>I've created this site to share some of my favorite recipes.</p>
```

2. A **blockquote** is a block-level element used for long quotations or quoted material that may consist of other block elements. The **q** (quote) element is for short quotations that go in the flow of text and do not cause line breaks.

3. The **pre** element.

4. The **ul** element is an unordered list element. It is used for lists that don't need to appear in a particular sequence. They display with bullets by default. The **ol** element is an ordered list in which sequence matters. The browser automatically inserts numbers for ordered lists.

5. Use a style sheet to remove bullets from an unordered list.

6. `<abbr title="World Wide Web Consortium">W3C</abbr>`

7. **dl** is the element used to identify an entire description list. The **dt** element is used to identify just one term within that list.

8. The **id** attribute is used to identify a unique element in a document, and the name in its value may appear only once in a document. **class** is used to classify multiple elements into conceptual groups.

9. An **article** element is intended for a self-contained body of content that would be appropriate for syndication or might appear in a different context. A **section** element divides content into thematically related chunks.

Chapter 6: Adding Links

1. `<a href="tutorial.html">...</a>`

2. `<a href="examples/instructions.html">...</a>`

3. `<a href="examples/people/family.html">...</a>`

4. `<a href="/examples/things/boot.html">...</a>`

5. `<a href="../index.html">...</a>`

6. `<a href="https://www.learningwebdesign.com">...</a>`

7. `<a href="../instructions.html">...</a>`

8. `<a href="../../index.html">...</a>`

9. `<img src="images/arrow.gif" alt="arrow pointing right" />`
10. `<img src="../images/arrow.gif" alt="arrow pointing right" />`
11. `<img src="../../images/bullet.gif" alt="small red dot" />`

Chapter 7: Adding Images

1. The **`src`** and **`alt`** attributes are required for the document to be valid. If the **`src`** attribute is omitted, the browser won't know which image to use. You may leave the value of the **`alt`** attribute empty for decorative images for which alt text would be meaningless or clumsy when read in context.
2. `<img src="furry.jpg" alt="">`
3. a) HTML documents are not valid if the **`alt`** attribute is omitted, and b) **`alt`** improves accessibility by providing a description of the image if it is not available or not viewable.
4. The three likely causes for a missing image are: a) the URL is incorrect, so the browser is looking in the wrong place or for the wrong filename (names are case-sensitive); b) the image file is not in an acceptable format; and c) the image file is not named with the proper suffix (*.gif*, *.jpg*, or *.png*, as appropriate).
5. The disk cache is where browsers temporarily store files downloaded over the network so they can be reused. Taking advantage of cached resources eliminates the need for repeated server requests for the same file and can increase performance.
6. Advantages include simple and familiar markup, excellent browser support, image caching, and available fallbacks. Disadvantages include the inability to manipulate the parts of the SVG with style sheets or JavaScript.
7. Advantages of inline SVGs include the ability to take advantage of all of SVG's features (animation, scripting, and applying CSS style rules), good browser support, and fewer server requests. Disadvantages include potentially unwieldy amounts of code in the HTML document, more complicated image maintenance, and lack of caching.
8. When it is purely presentational and not part of the editorial content of the page.
9. **`image/svg+xml`** is the MIME type of an SVG file. You may need to include it as the value of the **`type`** attribute in the **`picture`** element or use it to configure your server to recognize SVG files as images.

Chapter 8: Table Markup

1. The table itself (**`table`**), rows (**`tr`**), header cells (**`th`**), data cells (**`td`**), and an optional caption (**`caption`**).

2. The `table` element can directly contain `tr`, `caption`, `colgroup`, `thead`, `tbody`, and `tfoot` elements.
3. The `tr` element can contain only `th` and `td` elements.
4. Use the `colgroup` and `col` elements if you want to include additional information about the structure of a table, to specify widths to speed up display, or to add certain style properties to a column of cells.
5. Add a `caption` to describe the table contents, and use the `scope="col"` attribute to associate each `th` with its respective `td`s. You could use `headers` instead, but it isn't as reliable and the table isn't complicated enough to require them.
6. a. The `caption` should be the first element inside the `table` element.
   b. There may not be any text directly in the `table` element; it must go in a `th` or `td`.
   c. The `th` elements must go inside the `tr` element.
   d. There is no `colspan` element; it should be a `td` with a `colspan` attribute.

Chapter 9: Forms

1. a. POST (because of security issues)
   b. POST (because it uses the file selection input type)
   c. GET (because you may want to bookmark search results)
   d. POST (because it is likely to have a lengthy text entry)
2. a. Drop-down menu: `<select>`
   b. Radio buttons: `<input type="radio">`
   c. `<textarea>`
   d. Eight checkboxes: `<input type="checkbox">`
   e. Scrolling menu: `<select multiple="multiple">`
   f. Text input with `tel` type: `<input type="tel">`
   g. Date input: `<input type="date">`
3. a. `checkbox` is not an element name; it is a value of the `type` attribute in the `input` element.
   b. The `option` element is not a void (empty) element. It should contain the value for each option (for example, `<option>Orange</option>`).
   c. The required `name` attribute is missing.
   d. The width and height of a text area are specified with the `cols` and `rows` attributes, respectively.

Chapter 10: Embedded Media

1. A nested browsing context works like a browser window inside another browser window. You can create one with an **`iframe`** element or an **`object`** element (bonus points if you got both).
2. The **`sandbox`** attribute allows developers to set limitations on what nested content can do and is important for security reasons.
3. a. video codec
   b. video codec
   c. audio codec
   d. container for video or audio
   e. audio codec
   f. container for video or audio
   g. audio container and codec
4. The **`poster`** attribute specifies an image that appears in the video player until the video is played.
5. A *.vtt* file is a text file in the WebVTT format that contains subtitles, captions, descriptions, chapter titles, and/or metadata synchronized to a video or audio file.
6. SVG is a vector format, and Canvas is pixel-based (raster).
7. **`strokeRect()`** and **`fill()`**

Chapter 11: Introducing Cascading Style Sheets

1. selector: **`blockquote`**; property: **`line-height`**; value: **`1.5`**; declaration: **`line-height: 1.5;`**
2. The paragraph text will be gray because when there are conflicting rules of identical weight, the last one listed in the style sheet will be used.
3. a. Use one rule with multiple declarations applied to the **`p`** element:

```
p {
  font-family: sans-serif;
  font-size: 1em;
  line-height: 1.2em;
}
```

   b. The semicolons are missing:

```
blockquote {
  font-size: 1em;
  line-height: 150%;
  color: gray;
}
```

c. There should not be curly braces around every declaration, only around the entire declaration block:

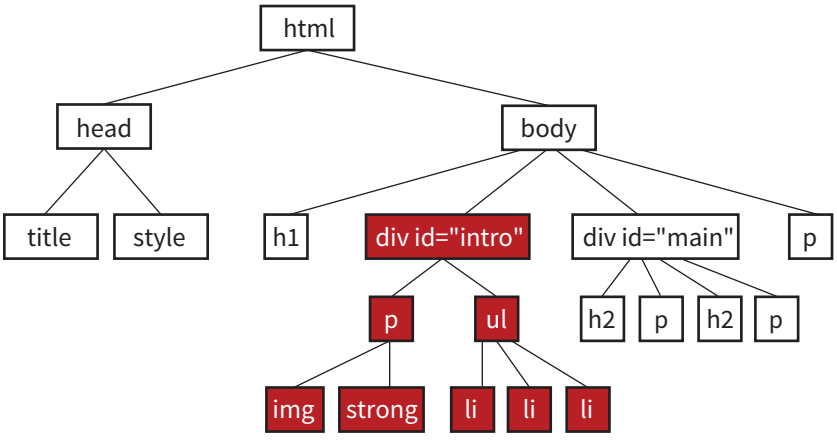
```
body {
  background-color: black;
  color: #666;
  margin-left: 12em;
  margin-right: 12em;
}
```

d. This could be handled with a single rule with a grouped element type selector:

```
p, blockquote, li {
  color: white;
}
```

e. This inline style is missing the property name:

```
<strong style="color: red">Act now!</strong>
```



NOTE: color styles applied to the `img` element are displayed on the border only (if there is one).

FIGURE A-1. Answer to Chapter 11, question 4)

Chapter 12: Formatting Text

1. a. 4; b. 1; c. 7; d. 3; e. 2; f. 9; g. 8; h. 5; i. 6

2. a. `body {color: red;}`

   b. `h2 {color: red;}`

   c. `h1, p {color: red;}`

   d. `.special {color: red;}`

   e. `#intro {color: red;}`

   f. `#main strong {color: red;}`

g. `h2 + p {color: red;}`

3. a. 0, 0, 2

   b. 0, 1, 1

   c. 0, 0, 2

   d. 1, 1, 2

   e. 0, 1, 2

4. d, because it has an ID selector and a 1 in the first box

Chapter 13: Colors and Backgrounds

1. a. 1; b. 3; c. 2; d. 6; e. 5; f. 4
2. Color model
3. Gamut
4. Color space
5. g
6. d (you can't use hexadecimal values in **`rgb()`** functions)

Chapter 14: Selectors

1. d
2. b
3. d
4. c
5. a. 5; b. 1; c. 4; d. 3; e. 2
6. d
7. b
8. d
9. c
10. Extra points if you included the **`text-transform`** property:

```
.quote::before {
  content: "Quote of the day: ";
  color: orange;
  text-transform: uppercase;
}
```

Chapter 15: Thinking Inside the Box

1. `border: double black medium;`
2. `padding: 2em;`
3. `padding: 2em; border: 4px solid red;`
4. `border: 4px solid red; margin: 2em;`
5. `padding: 1em 1em 1em 6em; border: 4px dashed black; margin: 1em 6em;`
   or:
   `padding: 1em; padding-left: 6em; border: 4px dashed; margin: 1em 6em;`
6. `padding: 1em 50px; border: 2px solid black; margin: 0 auto;`

Chapter 16: Floating and Positioning

1. b (floats are positioned against the content edge, not the padding edge)
2. c (floats do not use offset properties, so there is no reason to include **`right`**)
3. Clear the **`footer`** to make it start below a floated sidebar on either side: **`footer {clear: both;}`**
4. a. absolute
   b. relative, absolute, fixed
   c. fixed
   d. relative, absolute, fixed
   e. static
   f. relative
   g. absolute, fixed
   h. relative, absolute, fixed
   i. relative

Chapter 17: Getting in Line with Flexbox

1. Turn its parent element into a flex container by setting **`display`** to **`flex`**. The child element becomes a flex item automatically with no extra code.
2. a. 2; b. 4; c.1; d. 3
3. The **`align-items`** property positions items relative to their flex line, while **`align-content`** distributes space around and within multiple lines. Both properties are applied to the container, and both are concerned with positioning along the cross axis.
4. a. 4; b. 1; c. 3; d. 2

5. a. 4; b. 3; c. 2; d. 5; e. 1

6. The style rules for displaying the items in the order shown in FIGURE 17-30 are:

```
.box6 {order: -1;}

.box1, .box2, .box3 {order: 1;}
```

Chapter 18: Grid Layout

1. The key difference between Grid Layout and Flexbox is that Grid Layout creates layouts in two dimensions—rows and columns—but Flexbox arranges elements on one axis.

   Similarities between Grid Layout and Flexbox include:

   — For both, making an element a container automatically turns its direct children into items.

   — They are both based on the language direction of the document.

   — Both can create whole-page layouts (although Grid Layout is better suited to the task).

   — You can change the order of the items with the **order** property.

   — They both use the Box Alignment Module for item and content alignment.

2. Here's the grid template for the layout shown in FIGURE 18-28:

```
grid-template-rows: 12em min-content 100px;
grid-template-columns: 300px 2fr 1fr;
```

   Using the **grid** shorthand property:

```
grid: 12em min-content 100px / 300px 2fr 1fr;
```

   (Note: **auto** could be used instead of **min-content** in both examples.)

3. a. E; b. D; c. C; d. A; e. B

4.
```
#gallery {
  column-gap: 1em;
}
```

5. a. 2; b. 4; c. 3; d. 1; e. 5

Chapter 19: Responsive Web Design

1. It sets the size of the viewport of the mobile browser to the same size as the screen.

2.
```
img { max-width: 100%; }
```

3.
```
@media screen and (min-width: 60em) {
  body {
    margin: 0 10%;
  }
}
```

4. In an **`@media`** rule in a style sheet, in **`@import`** rules that call in external style sheets, and in the **`link`** element to an external style sheet.

5. It's an example of a fluid typography style rule. It sets the minimum size of the **`h1`** to .375rem, sets the maximum size to 1.875rem, and allows it to resize smoothly in between.

6. You could use an accordion to hide and reveal the submenu options, or put the submenu on a separate landing page and link to it from the main navigation.

7. **`#wrapper { container-type: inline-size; }`**

8.

```
@container (min-width: 800px) {
  ul {
    display: flex;
  }
}
```

9. Test on real devices, use simulator or emulator software, or use a third-party testing service.

10. A style guide documents the features of a brand identity (visual or written tone). It is generally one part of a greater design system that might also include component code libraries, pattern libraries, design tokens, and more.

11. A component is one small element in a user interface, such as a button. A pattern is a collection of components used together consistently across the site, such as a product listing that includes an image, heading, text, and a button.

12. Responsive typography changes based on breakpoints set in media queries. Fluid typography sizes gradually between specified minimum and maximum sizes.

Chapter 20: Transitions, Transforms, and Animation

1. Tweening is the process in animation in which frames are generated between two end states.

2. A transition would have two keyframes, one for the beginning state and one for the end.

3. a. `transition-delay: .5s;`

   b. `transition-timing-function: linear;`

   c. `transition-duration: .5s;`

   d. `transition-property: line-height;`

4. c. **`text-transform`** is not an animatable property.

5. **ease** is the default timing function. It starts out slowly, speeds up quickly, and then slows down again at the very end.
6. .2s is the **transition-duration** value (how long the animation lasts).
7. Trick question! They will arrive at the same time, 300 ms after the transition begins. The timing function has no effect on the total amount of time it takes.
8. a. `transform: rotate(7deg); or rotate: 7deg;`

   b. `transform: translate(-50px, -25px);` or `translate: -50px -25px;`

   c. `transform-origin: right bottom;`

   d. `transform: scale(1.25);` or `scale: 1.25;`
9. It indicates that the element should be resized three times larger than its original height.
10. The border is 3 pixels wide at 50% through the animation.
11. a. `animation-direction: reverse;`

    b. `animation-duration: 5s;`

    c. `animation-delay: 2s;`

    d. `animation-iteration-count: 3;`

Chapter 21: More CSS Techniques

1. d
2. Use Flexbox or floats.
3. The name of a custom property to be used as a CSS variable.
4. When a value is used repeatedly in a style sheet.
5.
```
div.card {
  padding: var(--spacing);
}
```
6. d
7. e
8. The **&** (ampersand) character indicates a nested selector.
9. Chrome, Safari, Firefox, and Edge, and their respective mobile versions.
10. (Note that there must be a character space after the **&** if a space is required after the preceding selector, such as **.options**.)
```
.options {
    font-weight: bold;
```

```
    & li {
      font-style: italic;
      margin-top: .5rem;

      & a {
        text-decoration: none;
        color: hotpink;
        border-bottom: 1px solid hotpink;

        &:hover,
        &:focus {
        color: forestgreen;
        border-color: forestgreen;
        }
      }
    }
}
```

11. Use an **`@supports`** rule:

```
@supports (display: grid) {
    /* #layout style rules here */
}
```

Chapter 22: Getting Started with JavaScript

1. c, a, b
2. `'<a href="https://oreilly.com">O'Reilly</a>'`
   `"<a href=\"https://oreilly.com\">O'Reilly</a>"`
3. **`console.log()`** prints the whatever is inside the parentheses to the JavaScript console.
4. b
5. d
6. c
7. d
8. d
9. a
10. d

Chapter 23: Functions, Conditionals, and Loops

1. b, a, c
2. d
3. a
4. b
5. 2, 3, 1

6. d
7. c
8. b
9. b
10. d

Chapter 24: Working with the DOM and Events

1. d
2. a
3. b and d
4. All of them
5. a and d
6. c, b, a
7. c
8. b
9. c and d
10. b, c, and d

Chapter 25: Next-Level JavaScript

1. d
2. a and c
3. c
4. Components
5. a
6. a
7. a, c, and d
8. b and d
9. b
10. c

Chapter 26: Web Image Basics

1. You can get a license to have exclusive rights to an image so that your competitor doesn't use the same photo on their site. You also know the source of the image is verified (i.e., it's not stolen).

2. "ppi" stands for "pixels per inch" and is a measure of resolution.
3. Create the image at twice the size it will appear in the layout to ensure it stays sharp on 2x displays.
4. Indexed color is a mode for storing pixel color information. GIF and PNG-8 are indexed color formats
5. WebP has transparency and can be used for animations.
6. WebP can combine alpha transparency with lossy compression, resulting in smaller files than PNG-24.
7. PNG offers alpha transparency.
8. Cumulative lossy compression means you lose image data every time you save an image as a JPEG. If you open a JPEG and save it as a JPEG again, even more image information is thrown out than the first time you saved it. Be sure to keep your full-quality original and save JPEG copies as needed.
9. In binary transparency, a pixel is either entirely transparent or entirely opaque. Alpha transparency allows up to 256 levels of transparency.
10. For many images, you could convert a JPEG to WebP, AVIF, or JPEG-XL for a smaller file size while maintaining the same quality.
11. Use the `picture` element to provide browsers with a list of image format options.
12. An ICO file is used for a favicon that appears in the browser tab or bookmarks menu. It goes in the root directory for the site.
13. c, d, and h
14. a, e, f, and g

Chapter 27: Image Production Techniques

1. CSS shapes and SVG format
2. You need to create multiple versions of each image.
3. d
4. d
5. e
6. By throwing away chunks of data dedicated to metadata and embedded color profiles.
7. A list of what image resources are available, and an indication of the size the `img` element will appear at in the layout.
8. `srcset`

9. e
10. To ensure they stay sharp on high-density displays.
11. Images get fuzzy when they are resized larger.

Chapter 28: SVG

1. `rect` is the SVG element that creates a rectangle.
2. The `x` and `y` coordinates position the rectangle element in the top-left corner of the SVG viewport.
3. The `width` and `height` attributes establish the dimensions of the SVG viewport, the area on which the drawing will be rendered.
4. In XML, all elements must be closed. When the element is a standalone element (without an opening and closing tag), it is closed with a forward slash (`/`) character before the closing bracket.
5. Clipping uses a vector shape to reveal or hide portions of an image. Masking is pixel-based, using the lightness and darkness of a raster image to hide and reveal the image.
6. Ways to reduce the size of an SVG include:
   - Simplifying paths
   - Reducing the number of decimal places
   - Using shapes instead of complex paths when possible
   - Avoiding raster images and effects
   - Running it through an optimizer like SVGO
   - Enabling Gzip or Brotli compression on the server
7. b (and for the record, SVG, Electric Boogaloo isn't a thing)
8. You can style SVGs in the following ways:
   - SVG presentation attributes
   - The inline `style` attribute
   - A style sheet in the SVG itself (`style` element)
   - An external style sheet called into the SVG (for SVGs placed with the `img` element)
   - If the SVG is inline, the style sheet in the HTML document in which it appears
9. c
10. `preserveAspectRatio="none"`
11. d

APPENDIX

B

HTML GLOBAL ATTRIBUTES

The global attributes listed here may be used with any HTML element. For more detailed information, please see:

- The HTML spec (*html.spec.whatwg.org/multipage/dom.html#global-attributes*)
- The "Global Attributes" overview on MDN Web Docs (*developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes*)

Attribute	Values	Description
accesskey	*single text character*	Assigns an access key (shortcut key command) to the link. Access keys are also used for form fields. Users may access the element by pressing Alt-<key> (PC) or Ctrl-<key> (Mac). Example: **accesskey="B"**.
autocapitalize	**none** | **off** | **on** | **sentences** | **words** | **characters**	Provides capitalization hints for elements where **contenteditable** is set to **true**, and for **form** and **input** elements to direct the capitalization of user-provided content (with the exception of the URL, email, and password input states, which never have capitalization enabled).
autocorrect	**on** | **off** | **""** (empty string)	Enables automatic correction of spelling and/or punctuation errors for elements where **contenteditable** is set to **true**, and for **form** and **input** elements (with the exception of the URL, email, and password input states). Experimental as of this writing.
autofocus	Boolean attribute; no values	Indicates the element should be focused when the page loads.
class	*text string*	Assigns one or more classification names to the element. Multiple values are separated by spaces.
contenteditable	**true** | **false** | plaintext-only | **""** (empty string)	Indicates whether the user can edit the element. An empty string (**""**) is the same as **true**. By default, an element inherits the setting from its parent. **plaintext-only** indicates the element's raw text may be edited, but rich text formatting is disabled.

Attribute	Values	Description
dir	**ltr** | **rtl** | **auto**	Specifies the inline text direction of the element ("left to right" or "right to left") and scopes bidirectional reordering, isolating the text from influencing surrounding content. When set to **auto**, the user agent decides based on the directionality of the first characters inside the element.
draggable	**true** | **false**	Indicates whether the element is draggable in the UI (an event configured with JavaScript), meaning the user can move it by clicking and holding it, and then moving it to a new position in the window.
enterkeyhint	**enter** | **done** | **go** | **previous** | **search** | **next**	Specifies how the enter key is labeled (or which icon should be shown) on a virtual keyboard, as an indication of what action takes place next.
hidden	Boolean attribute; no values	Prevents the element and its descendants from being rendered in the user agent (browser). Any scripts or form controls in hidden sections will still execute but will not be presented to the user.
id	*text string* (may not contain spaces)	Assigns a unique identifying name to the element. The value may not contain spaces.
inert	Boolean attribute; no values	Specifies content that should not be interactive. When an element is inert, normally interactive buttons such as links and form controls are disabled.
inputmode	**no** | **text** | **decimal** | **numeric** | **tel** | **search** | **email** | **url**	Provides a hint about the type of content the user may enter so the browser can display an appropriate virtual keyboard.
is	*text string* (references predefined custom element name)	Indicates that a standard HTML element should behave like a custom built-in element. (Poorly supported and may be deprecated.)
itemid	*URL* or *URN*	Provides the unique global identifier (in the form of a URL or URN) for an item. **itemid** is part of a system for providing microdata and may be used only with elements that have both **itemscope** and **itemtype** specified.
itemprop	*text string*	Used to add properties to an element, creating name/value pairs. For example: `<li itemprop="author">Jennifer Robbins</li>` results in the name/value pair author: Jennifer Robbins.
itemref	*id value*	Indicates the element is associated with another element on the page (identified by its **id**) without it needing to be a descendant of that element or breaking the tree structure of the document.
itemscope	Boolean attribute; no values	Defines the scope of associated metadata when working with structured data in HTML. When **itemscope** is applied to an element, it creates a new "item," resulting in a number of name/value pairs associated with that item.
itemtype	*URL*	Specifies the URL of the structured data vocabulary used to define item properties (**itemprop**).
lang	ISO language code (see ***loc.gov/standards/iso639-2/php/code_list.php***)	Specifies the primary language for the content of an element and its attribute values. When **lang** is omitted, the language of the element is the same as that of its parent element.

Attribute	Values	Description
nonce	*text string*	Defines a cryptographic "number used once" (nonce) as part of a Content Security Policy controlling whether a fetch will be allowed to proceed for a particular element.
popover	Boolean attribute; no values	Controls "popover" functionality. When **popover** is applied, the element won't be rendered until it is invoked by another element, at which point it will be rendered on top of other page content.
spellcheck	**true** | **false**	Indicates whether the element is to have its spelling and grammar checked. When omitted, the element follows the default behavior for that element, possibly inheriting the parent's **spellcheck** state.
style	Semicolon-separated list of style rules (*property: value pairs*)	Associates style information with an element. For example: `<h1 style="color: red; border: 1px solid">Heading</h1>`
tabindex	*number*	Indicates that the element is focusable and specifies its position in the tabbing order for the current document. The value must be between **0** and **32767**. It is used for tabbing through links on a page or fields in a form and is useful for assistive browsing devices. A value of **-1** is allowable to remove elements from the tabbing flow and make them focusable only by JavaScript.
title	*text string*	Provides a title or advisory information about the element, typically displayed as a tooltip. If a title is not specified, an element inherits the title from its nearest ancestor element with a title.
translate	**yes** | **no**	Specifies whether or not to translate an element's attribute values and the values of its text node children when the page is localized. If not specified, the element inherits the **translate** state from its parent.
writingsuggestions	**true** | **false**	Indicates whether browser-provided writing suggestions should be enabled. (Poorly supported.)

APPENDIX

C

CSS SELECTORS, LEVEL 4

The following table lists the selectors in the Selectors Level 4 spec (*w3.org/TR/selectors-4/*, November 2022).

CSS Selectors, Level 5

The W3C is working on the Level 5 spec (*drafts.csswg.org/selectors-5/*), which is an Editor's Draft as of this writing. It adds the `:local-link` functional pseudo-class selector to target links based on URL path structure, the `:state()` pseudo-class selector for matching custom elements, and a reference combinator, `/ref/`, for creating complex compound selectors. Although `:state()` is newly available in modern browsers, the other selector types are considered experimental as of this writing.

Selector	Type of selector	Description
Elemental selectors		
*	Universal selector	Matches any element.
E	Type selector	Matches the name of an element.
E:defined	Defined pseudo-class	Matches an element based on whether it is defined in the host language (such as custom elements).
Combinators		
E, F	Compound selector	Matches elements E and F.
E F	Descendant combinator	Matches element F only if it is a descendant of element E.
E > F	Child combinator	Matches any element F that is a child of element E.
E + F	Next-sibling combinator	Matches any element F that immediately follows any element E, where E and F share the same parent.
E ~ F	Subsequent-sibling combinator	Matches any element F that is preceded by E, where E and F share the same parent.

Selector	Type of selector	Description
Class and ID selectors		
`.classname` `E.classname`	Class selector	Matches the value of the class attribute in all elements or in a specified element.
`#idname` `E#idname`	ID selector	Matches the value of the **`id`** attribute in an element.
Attribute selectors		
`E[att]`	Simple attribute selector	Matches any element E that has the given attribute defined, whatever its value.
`E[att="val"]`	Exact attribute value	Matches any element E that has the specified attribute set to the specified value.
`E[att="val" i]`	Case-insensitive attribute value	Matches any element E that has the specified attribute set to the specified value, even if it does not match its capitalization (including in XML languages that may be case-sensitive). This example matches images named Icon.png, ICON.png, icon.png, and so on: `img[src="Icon.png" i]`
`E[att="val" s]`	Case-sensitive attribute value	Matches any element E that has the specified attribute set to the exact value provided, including matching capitalization.
`E[att~="val"]`	Partial attribute value	Matches any element E that has the specified value as one of the values in a list given to the specified attribute. For example: `table[class~="example"]`
`E[att|="val"]`	Hyphenated prefix attribute value	Matches any element E that has the specified attribute with a value that is equal to the provided string or begins with that string followed by a dash (-). Most often used to select languages, as shown here: `a[lang|="en"]`
`E[att^="val"]`	Beginning substring attribute	Matches any element E that has the specified attribute, and its value begins with the provided string. For example: `img[src^="/images/icons"]`
`E[att$="val"]`	Ending substring attribute	Matches any element E that has the specified attribute, and its value ends with the provided string. For example: `img[src$=".svg"]`
`E[att*="val"]`	Arbitrary substring attribute	Matches any element E that has the specified attribute, and its value contains the provided string. For example: `img[title*="July"]`
Location pseudo-class selectors		
`:any-link`	Hyperlink pseudo-class	Selects a link regardless of whether it has been visited.
`:link`	Link history pseudo-class	Selects links that have not yet been visited.
`:target`	Target pseudo-class	Selects an element that is used as a fragment identifier.
`:target-within`	Target container pseudo-class	Selects an element that is used as a fragment identifier or contains an element that is.
`:visited`	Link history pseudo-class	Selects links that have already been visited.

Selector	Type of selector	Description
`:local-link`	Local link pseudo-class	Selects links within the current site (i.e., with absolute URLs that match that of the current site).
`E:scope`	Reference element pseudo-class	Matches an element E only if it appears within a scoping root (a particular subtree or fragment of the document).
User action pseudo-class selectors		
`:active`	User action pseudo-class	Selects any element that has been activated by the user, such as a link as it is being clicked.
`:hover`	User action pseudo-class	Selects elements (typically links) that appear when the cursor is placed over them.
`:focus`	User action pseudo-class	Selects any element that currently has input focus, such as a selected form input.
`:focus-within`	Focus container pseudo-class	Selects any element that has or contains an element that has user input focus.
`:focus-visible`	Focus-indicated pseudo-class	Selects any element that has input focus and for which the user agent has determined that a focus ring or other indicator should be drawn.
Language-based selectors		
`:dir(ltr)`	Directionality pseudo-class	Selects an element with a particular writing direction (**`ltr`**, as shown here, for left to right, or **`rtl`** for right to left). How directionality is determined is dependent on the document language.
`:lang(xx)`	Language pseudo-class	Selects an element that matches the provided two-character language code. For example: `a:lang(de)`
Tree-structural pseudo-class selectors		
`:root`	Tree-structural pseudo-class	Selects an element that is the root of the document. In HTML, this is the html element.
`:empty`	Tree-structural pseudo-class	Selects an element that has no child elements and no content except maybe whitespace.
`:nth-child(n)`	Child-indexed pseudo-class	Selects an element that is the nth child of its parent. The notation can include a number, a notation in the An+B format, or the keywords odd or even.
`:nth-last-child(n)`	Child-indexed pseudo-class	Selects an element that is the nth child of its parent, counting backward from the last one.
`:first-child`	Child-indexed pseudo-class	Selects an element that is the first child of its parent.
`:last-child`	Child-indexed pseudo-class	Selects an element that is the last child of its parent.
`:only-child`	Child-indexed pseudo-class	Selects an element that is the only child of its parent.
`:nth-of-type(n)`	Typed child-indexed pseudo-class	Selects the nth element of its type.
`:nth-last-of-type(n)`	Typed child-indexed pseudo-class	Selects the nth element of its type, counting backward from the last one.
`:first-of-type`	Typed child-indexed pseudo-class	Selects an element that is the first sibling of its type.

Selector	Type of selector	Description
`:last-of-type`	Typed child-indexed pseudo-class	Selects an element that is the last sibling of its type.
`:only-of-type`	Typed child-indexed pseudo-class	Selects an element that is the only sibling of its type.
Input pseudo-class selectors		
`:enabled`	Enabled input pseudo-class	Selects a UI element if it is enabled.
`:disabled`	Disabled input pseudo-class	Selects a UI element if it is disabled.
`:read-write`	Mutability pseudo-class	Selects a UI element if it is user-alterable.
`:read-only`	Mutability pseudo-class	Selects a UI element if it is not user-alterable.
`:placeholder-shown`	Placeholder-shown pseudo-class	Selects an input control currently showing placeholder text.
`:autofill`	Automatic input pseudo-class	Selects inputs automatically filled by the user agent (not altered by the user).
`:default`	Default-option pseudo-class	Selects a UI element that is the default item in a group of related choices.
`:checked`	Selected-option pseudo-class	Selects a UI element (radio button or checkbox) that is checked.
`:indeterminate`	Indeterminate-value pseudo-class	Selects a UI element that is in an indeterminate state (neither checked nor unchecked).
`:blank`	Empty-value pseudo-class	Selects an element that has no content except maybe whitespace.
`:valid`	Validity pseudo-class	Selects a UI element that meets its data validity semantics.
`:invalid`	Validity pseudo-class	Selects a UI element that does not meet its data validity semantics.
`:in-range`	Range pseudo-class	Selects a UI element whose current value is in the range specified by the min and max attributes.
`:out-of-range`	Range pseudo-class	Selects a UI element whose value is not in the range specified by the min and max attributes.
`:required`	Optionality pseudo-class	Selects a UI element that requires input.
`:optional`	Optionality pseudo-class	Selects a UI element that does not require input.
`:user-valid`	User-interaction pseudo-class	Selects a UI element with correct user input.
`:user-invalid`	User-interaction pseudo-class	Selects a UI element with incorrect user input.
Logical combination pseudo-class selectors		
`E:not(A, B, ...)`	Negation pseudo-class	Selects an element that does not match any of the provided selectors.
`E:is(A, B, ...)`	Matches-any pseudo-class	Selects an element that matches one or more of the provided selectors.
`E:where(A, B, ...)`	Specificity-adjustment pseudo-class	Selects an element that matches one or more of the provided selectors but contributes no specificity.
`E:has(rA, rB, ...)`	Relational pseudo-class	Selects an element when any of the provided relative selectors match at least one element. Provides a method for selecting an element relative to a reference element, such as selecting a parent element or a previous sibling. The following example matches only a elements that contain an **`img`** element: `a:has(> img)`

Selector	Type of selector	Description
Time-dimensional pseudo-class selectors		
`:current`	Current-element pseudo-class	Selects an element that is currently being presented in a timeline (for example, being read by a screen reader during the display of a video with subtitles). This feature is experimental as of this writing.
`:current(A, B...)`	Current-element pseudo-class	Selects the deepest :current element that matches the provided selectors. This feature is experimental as of this writing.
`:past`	Past-element pseudo-class	Selects an element that is in the past in the timeline.
`:future`	Future-element pseudo-class	Selects an element that is in the future in the timeline.
Grid-structural selectors		
`F || E`	Column combinator	Selects an element E that represents a cell in a grid or table belonging to a column represented by an element F.
`:nth-col(n)`	Column pseudo-class	Selects an element that represents a cell belonging to the nth column in a grid or table.
`:nth-last-col(n)`	Column pseudo-class	Selects an element that represents a cell belonging to the nth column in a grid or table, counting backward from the last one.
Pseudo-element selectors		
`::first-letter`	Pseudo-element selector	Selects the first letter of the specified element.
`::first-line`	Pseudo-element selector	Selects the first line of the specified element.
`::before`	Pseudo-element selector	Inserts generated text at the beginning of the specified element and applies a style to it.
`::after`	Pseudo-element selector	Inserts generated content at the end of the specified element and applies a style to it.

APPENDIX

D

SHORTENED URL REFERENCE

URLs that are very long, feature long strings of random characters, or are generally tricky to type have been shortened using O'Reilly's URL shortener. The table below lists the full URL for each shortened version in the book.

Shortened URL	Complete URL
Chapter 1, Teams and Tools	
oreil.ly/laaJt	bradfrost.com/blog/post/front-of-the-front-end-and-back-of-the-front-end-web-development
Chapter 4, Creating a Simple Page	
oreil.ly/99esV	iana.org/assignments/language-subtag-registry/language-subtag-registry
Chapter 8, Table Markup	
oreil.ly/aWN0l	html.spec.whatwg.org/multipage/tables.html#tables
Chapter 10, Embedded Media	
oreil.ly/M3AR1	developer.mozilla.org/docs/Learn/HTML/Multimedia_and_embedding/Other_embedding_technologies
oreil.ly/H1RFo	developer.mozilla.org/Apps/Fundamentals/Audio_and_video_delivery/cross_browser_video_player
oreil.ly/82gFf	cloudinary.com/blog/build-a-custom-html5-video-player-with-javascript
oreil.ly/IsL_9	developer.mozilla.org/Apps/Fundamentals/Audio_and_video_delivery/Adding_captions_and_subtitles_to_HTML5_video
oreil.ly/J6wa8	iana.org/assignments/language-subtag-registry/language-subtag-registry
oreil.ly/efId4	code.tutsplus.com/articles/21-ridiculously-impressive-html5-canvas-experiments--net-14210
Chapter 11, Introducing Cascading Style Sheets	
oreil.ly/wt3Mm	learn.microsoft.com/microsoft-edge/devtools-guide-chromium/landing/

Shortened URL	Complete URL
Chapter 12, Formatting Text	
oreil.ly/f-TEC	developer.mozilla.org/docs/Web/CSS/revert
oreil.ly/QP2bS	w3.org/TR/css-cascade-4/#defaulting-keywords
oreil.ly/kJ_-n	developer.mozilla.org/docs/Learn/CSS/Styling_text/Web_fonts
oreil.ly/1p6OF	developer.mozilla.org/docs/Web/CSS/CSS_fonts/Variable_fonts_guide
oreil.ly/K7KrU	fonts.google.com/knowledge/introducing_type/introducing_variable_fonts
oreil.ly/M48jW	practice.typekit.com/lesson/caring-about-opentype-features/
oreil.ly/emg3P	developer.mozilla.org/docs/Web/CSS/Specificity
oreil.ly/icCTv	developer.mozilla.org/docs/Web/CSS/::marker
Chapter 13, Colors and Backgrounds	
oreil.ly/pHsBE	developer.chrome.com/articles/high-definition-css-color-guide/
oreil.ly/g1pdq	webkit.org/blog/6682/improving-color-on-the-web/
oreil.ly/aTVLI	keithjgrant.com/posts/2023/04/its-time-to-learn-oklch-color/
oreil.ly/49cmu	developer.mozilla.org/docs/Web/CSS/gradient/conic-gradient
Chapter 14, Selectors	
oreil.ly/azcov	smashingmagazine.com/2011/03/how-to-use-css3-pseudo-classes/
Chapter 15. Thinking Inside the Box (the CSS Box Model)	
oreil.ly/FMmG7	css-tricks.com/inheriting-box-sizing-probably-slightly-better-best-practice
oreil.ly/E3Wtv	developer.mozilla.org/docs/Web/CSS/display/multi-keyword_syntax_of_display
oreil.ly/61NTE	complexspiral.com/publications/uncollapsing-margins
oreil.ly/F5VeG	css-tricks.com/what-you-should-know-about-collapsing-margins/
oreil.ly/dhaDy	w3.org/TR/css-backgrounds-3/#border-images
oreil.ly/7vQEt	css-tricks.com/almanac/properties/b/border-image/
Chapter 16, Floating and Positioning	
oreil.ly/kLkLj	firefox-source-docs.mozilla.org/devtools-user/page_inspector/how_to/edit_css_shapes/index.html
Chapter 17, Getting in Line with Flexbox	
oreil.ly/WHVhe	developer.mozilla.org/docs/Learn/CSS/CSS_layout/Flexbox
oreil.ly/a5Qx8	developer.mozilla.org/docs/Web/CSS/CSS_flexible_box_layout
oreil.ly/YhPJO	codepen.io/enxaneta/full/adLPwv/
Chapter 18, Grid Layout	
oreil.ly/7n8Xr	developer.mozilla.org/docs/Web/CSS/CSS_grid_layout/Subgrid
oreil.ly/NkaOb	css-tricks.com/auto-sizing-columns-css-grid-auto-fill-vs-auto-fit/
oreil.ly/bh7BD	developer.mozilla.org/docs/Web/CSS/CSS_grid_layout
oreil.ly/_uDUf	developer.mozilla.org/docs/Learn/CSS/CSS_layout/Grids

Shortened URL	Complete URL
Chapter 19, Responsive Web Design	
oreil.ly/LevHr	youtube.com/watch?v=AMPKmh98XLY
oreil.ly/xz-Q-	smashingmagazine.com/2020/09/design-reduced-motion-sensitivities/
oreil.ly/nDR56	smashingmagazine.com/2023/08/css-accessibility-inclusion-user-choice/
oreil.ly/h4SAH	mgearon.com/css/user-preference-media-features/
oreil.ly/vGvgb	developer.mozilla.org/blog/getting-started-with-css-container-queries/
oreil.ly/xgdUB	web.dev/blog/how-to-use-container-queries-now
oreil.ly/KPZGP	smashingmagazine.com/2021/05/complete-guide-css-container-queries/
oreil.ly/gBJFl	youtube.com/watch?v=3_-Je5XpbqY
oreil.ly/vgfi_	lukew.com/ff/entry.asp?1514
oreil.ly/cNHzX	bradfrost.com/blog/post/atomic-web-design/
oreil.ly/2QdIW	bigmedium.com/ideas/the-art-of-design-system-recipes.html
oreil.ly/AnR7Q	github.com/sturobson/Awesome-Design-Tokens
oreil.ly/BY_sz	medium.com/@nitishkmrk/design-tokens-beginners-guide-4424944bf5f9
oreil.ly/GaTDt	bradfrost.com/blog/post/creating-themeable-design-systems/
oreil.ly/XMSZk	help.figma.com/hc/en-us/sections/14548397990423-Course-Introduction-to-design-systems
oreil.ly/gV05K	bradfrost.github.io/this-is-responsive/patterns.html
oreil.ly/ZjwC7	designshack.net/articles/ux-design/5-really-useful-responsive-web-design-patterns/
oreil.ly/zhZm4	smashingmagazine.com/2023/11/addressing-accessibility-concerns-fluid-type
oreil.ly/25c3m	developer.mozilla.org/docs/Web/CSS/CSS_Functions/Using_CSS_math_functions
oreil.ly/VYLsa	smashingmagazine.com/2022/01/modern-fluid-typography-css-clamp/
oreil.ly/PdmQs	blog.logrocket.com/fluid-vs-responsive-typography-css-clamp/
oreil.ly/X4G0n	bradfrost.com/blog/web/responsive-nav-patterns
oreil.ly/X-435	bradfrost.com/blog/post/complex-navigation-patterns-for-responsive-design/
oreil.ly/Y6-vp	bradfrost.github.io/this-is-responsive/patterns.html
oreil.ly/VCgrW	smashingmagazine.com/2022/11/navigation-design-mobile-ux/
oreil.ly/CElqB	smashingmagazine.com/2022/03/guide-hover-pointer-media-queries/
oreil.ly/kRr9D	bigmedium.com/ideas/designing-for-touch-book.html
oreil.ly/0ile3	alistapart.com/articles/creating-intrinsic-ratios-for-video
oreil.ly/wUlUG	dev.to/peboy/how-to-create-a-responsive-video-background-in-html-and-css-1bh6
oreil.ly/Q7Dry	smashingmagazine.com/2022/12/accessible-front-end-patterns-responsive-tables-part1/
oreil.ly/PFZ44	smashingmagazine.com/2022/12/accessible-front-end-patterns-responsive-tables-part2/
oreil.ly/5xBcO	css-tricks.com/accessible-simple-responsive-tables/
oreil.ly/SUiwc	bradfrost.github.io/this-is-responsive/resources.html
oreil.ly/6aHCv	frontenddogma.com/topics/responsive-design/

Shortened URL	Complete URL
Chapter 20, Transitions, Transforms, and Animation	
oreil.ly/zFU6F	css-tricks.com/using-multi-step-animations-transitions/
oreil.ly/ZkV00	coding.smashingmagazine.com/2012/01/06/adventures-in-the-third-dimension-css-3-d-transforms
oreil.ly/9i_ap	robots.thoughtbot.com/css-animation-for-beginners
oreil.ly/k8qHE	smashingmagazine.com/2011/09/the-guide-to-css-animation-principles-and-examples/
oreil.ly/nwywz	developers.google.com/web/tools/chrome-devtools/inspect-styles/animations
oreil.ly/KTwhf	webkit.org/blog/10266/web-animations-in-safari-13-1/
oreil.ly/nZSP5	developer.mozilla.org/docs/Tools/Page_Inspector/How_to/Work_with_animations
oreil.ly/pCk0b	learn.microsoft.com/en-us/microsoft-edge/devtools-guide-chromium/inspect-styles/animations
oreil.ly/PT1ff	tympanus.net/codrops/2011/11/02/original-hover-effects-with-css3/
Chapter 21, More CSS Techniques	
oreil.ly/Z1qs2	css-tricks.com/zero-trickery-custom-radios-and-checkboxes/
oreil.ly/Yg3QK	moderncss.dev/pure-css-custom-styled-radio-buttons/
oreil.ly/Ijiui	moderncss.dev/pure-css-custom-checkbox-style/
oreil.ly/qlt9h	blog.logrocket.com/creating-custom-select-dropdown-css/
oreil.ly/O0bHv	github.com/sindresorhus/modern-normalize/releases/tag/v2.0.0
oreil.ly/pXM1r	codepen.io/zachwolf/details/bdZMZj
oreil.ly/KIG_K	microsoft.com/en-us/edge/download/insider
oreil.ly/nk_AY	developer.apple.com/safari/technology-preview/
oreil.ly/fR9hc	mozilla.org/en-US/firefox/channel/desktop/
oreil.ly/NE5rP	webkit.org/blog/14955/the-web-just-gets-better-with-interop/
oreil.ly/zMPrG	developer.mozilla.org/docs/Web/CSS/@supports
oreil.ly/NuuqW	w3.org/TR/css-properties-values-api-1/#at-property-rule
oreil.ly/nuj36	smashingmagazine.com/2024/05/times-need-custom-property-instead-css-variable/
oreil.ly/VA895	css-tricks.com/a-complete-guide-to-custom-properties
oreil.ly/UzhXm	developer.mozilla.org/docs/Web/CSS/Using_CSS_custom_properties
oreil.ly/FJRwd	blog.logrocket.com/how-to-use-css-variables-like-a-pro/
oreil.ly/y_nt6	developer.mozilla.org/docs/Web/CSS/CSS_nesting/Using_CSS_nesting
oreil.ly/ki0XO	youtube.com/watch?v=YnWPeA6l5UE
oreil.ly/F1-Jt	developer.mozilla.org/docs/Learn/CSS/Building_blocks/Cascade_layers
Chapter 22, Getting Started with JavaScript	
oreil.ly/MxXls	developer.mozilla.org/docs/Learn_web_development/Core/Scripting/Object_basics
oreil.ly/U193I	developer.mozilla.org/docs/Learn/JavaScript/First_steps/Math
oreil.ly/Op9SE	freecodecamp.org/news/template-literals-in-javascript/
oreil.ly/egK8A	developer.mozilla.org/docs/Web/JavaScript/Reference/Regular_expressions/Character_escape
oreil.ly/9ofvq	geeksforgeeks.org/what-is-type-coercion-in-javascript/

Shortened URL	Complete URL
Chapter 22, Continued	
oreil.ly/tMLTU	developer.mozilla.org/docs/Web/JavaScript/Reference/Global_Objects/String
oreil.ly/jzFHt	developer.mozilla.org/docs/Web/JavaScript/Guide/Expressions_and_operators
oreil.ly/jbLWi	developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/var
Chapter 23, Functions, Conditionals, and Loops	
oreil.ly/HbBm3	freecodecamp.org/news/scope-in-javascript-global-vs-local-vs-block-scope/
oreil.ly/-mm-T	developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/if...else
oreil.ly/vLY3W	medium.com/@rabailzaheer/truthy-falsy-values-and-equality-operator-a-javascript-guide-3a0760d85cc8
oreil.ly/R1sk5	digitalocean.com/community/tutorials/understanding-arrays-in-javascript
oreil.ly/Rv6vh	youtube.com/watch?v=Kn06785pkJg
Chapter 24, Working with the DOM and Events	
oreil.ly/GQeY1	kryogenix.org/code/browser/everyonehasjs.html
oreil.ly/HVfCb	developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
oreil.ly/_Wcdb	freecodecamp.org/news/innerhtml-vs-innertext-vs-textcontent/
oreil.ly/lQgOg	smashingmagazine.com/2013/11/an-introduction-to-dom-events/?
oreil.ly/wSwWK	www.aleksandrhovhannisyan.com/blog/interactive-guide-to-javascript-events
Chapter 25, Next-Level JavaScript	
oreil.ly/wd1ls	developer.mozilla.org/ /docs/Learn_web_development/Core/Frameworks_libraries
oreil.ly/-16EC	developer.mozilla.org/docs/Web/JavaScript/Guide/Working_with_objects
oreil.ly/w_IOe	eisenbergeffect.medium.com/about-web-components-7b2a3ed67a78
oreil.ly/FLMSA	freecodecamp.org/news/build-your-first-web-component/
oreil.ly/smhUZ	timkadlec.com/remembers/2020-04-21-the-cost-of-javascript-frameworks/
oreil.ly/hB-Hw	2024.stateofjs.com/libraries/front-end-frameworks/
oreil.ly/ZUE4c	vueschool.io/lessons/getting-started-with-vue-js-and-the-composition-api
oreil.ly/1sGYB	freecodecamp.org/news/get-started-with-nodejs/
oreil.ly/KFXYQ	alistapart.com/article/yes-that-web-project-should-be-a-pwa/
Chapter 26, Web Image Basics	
oreil.ly/lNOx7	medium.com/@duhroach/how-png-works-f1174e3cc7b7
oreil.ly/WLmyM	creativecloud.adobe.com/learn/app/photoshop
oreil.ly/Hx0w6	adobe.com/products/photoshop/free-trial-download.html
oreil.ly/NcW3F	chromium.googlesource.com/codecs/libwebp2/
oreil.ly/rn4dw	figma.com/community/plugin/1181873200384736932/webp-exporter
oreil.ly/OyPhl	aomedia.org/blog%20posts/avif-meet-the-next-level-image-file-format/
oreil.ly/k19Hw	netflixtechblog.com/avif-for-next-generation-image-coding-b1d75675fe4
oreil.ly/iekl7	opensource.googleblog.com/2024/04/introducing-jpegli-new-jpeg-coding-library.html
oreil.ly/irOfE	medium.com/@pnowelldesign/pixel-density-demystified-a4db63ba2922

Shortened URL	Complete URL
Chapter 26, Continued	
oreil.ly/zmABc	sebastien-gabriel.com/designers-guide-to-dpi/
oreil.ly/vSg_5	evilmartians.com/chronicles/how-to-favicon-in-2021-six-files-that-fit-most-needs
oreil.ly/LAs5L	smashingmagazine.com/2016/05/easy-steps-to-better-logo-design
oreil.ly/bEmZB	careerfoundry.com/en/blog/ui-design/icon-design-process/
Chapter 27, Image Production Techniques	
oreil.ly/SqETB	web.dev/articles/optimize-css-background-images-with-media-queries
oreil.ly/80DCp	medium.com/web-dev-survey-from-kyoto/responsive-images-diy-implementation-in-6-steps-a4342ecbb08
oreil.ly/uZ0lO	cloudinary.com/blog/introducing_intelligent_responsive_image_breakpoints_solutions
oreil.ly/KN0yd	smashingmagazine.com/2016/01/responsive-image-breakpoints-generation/
oreil.ly/n-54l	cloudfour.com/thinks/responsive-images-101-part-9-image-breakpoints/
oreil.ly/i6Yc0	imagekit.io/blog/what-is-content-delivery-network-cdn-guide/
oreil.ly/ENKTP	imagekit.io/blog/what-is-image-cdn-guide/
Chapter 28, SVG	
oreil.ly/D9VHO	medium.com/@hayavuk/complete-guide-to-svg-sprites-7e202e215d34
oreil.ly/XVIJ9	smashingmagazine.com/2021/05/accessible-svg-patterns-comparison/
oreil.ly/oVcNL	sarasoueidan.com/blog/svg-coordinate-systems/
oreil.ly/mny_V	tympanus.net/codrops/2014/08/19/making-svgs-responsive-with-css/

INDEX

Symbols

` (backticks), string template literals 618
++ (plus plus), incrementing operator 616
+= (plus equals), addition assignment operator 616, 674
+ (plus)
  addition operator 613
  next-sibling selector 271
  :nth-child selector 341
<= less than or equal to operator 614
< > (angle brackets), HTML tags 58
<!-- and -->, comment tags 57, 63
< less than operator 614
=== (triple equals sign), equality comparison 615
== (double equals), equivalent to operator 614, 615
*= (asterisk and equals), multiplication assignment operator 616
= (equals)
  assignment operator 615
  attribute values 66
-= (hyphen equals), subtraction assignment operator 616
!= not equal to operator 614
/= (slash equals) division assignment operator 616
>= greater than or equal to operator 614
> (greater than)
  child selector 271
  JavaScript comparison operator 614
|| (double bars), column combinators 271
| (single bar), logical OR operator 648
~ (tilde), subsequent-sibling selector 271
& (ampersand)
  character references 107
  nested selectors 597
* (asterisk)
  multiplication operator 614
  universal element selector 272
\ (backslash)
  avoiding in URL pathnames 118
  versus slash 58
{ } (curly braces)
  declaration block 228
  nesting consideration 513
&& (double ampersand), logical AND operator 648
:: (double colon), pseudo-element selectors 348
{{ }} (double curly braces), template properties 709
-- (double hyphen), decrementing operator 616
" (double quotes)
  strings 618, 619
  wrapping attribute values in 66
// (double slashes), comments 609
! (exclamation mark), logical NOT operator 649
- (hyphen)
  custom element names 699
  subtraction operator 613
!important indicator, assigning to style 236, 272
(octothorpe symbol)
  for hexadecimal color values 300
  ID selectors 271
() (parentheses), in JavaScript 636, 681
% (percent), image positioning 419
. (period)
  class selectors 272
  dot notation 611
; (semicolon)
  character references 107
  CSS declarations 229
  statements 609, 612
' (single quotes), JavaScript strings 618
/ (slash)
  versus backslash 58
  division operator 614
  Unix pathname convention 124
[ ] (square brackets), grid templates 478
_ (underscore), in variable names 623

A

A11y 43. *See also* accessibility
AAC audio codec 209, 210
abbr element 95
absolute flex 450, 451
absolute path 28
absolute positioning 401, 411, 414–418
absolute units, CSS 241–242
absolute URLs 116
accent-color property 581
accept attribute, file input type 188

accessibility 25, 43–44
  alternative text for images 139
  ARIA attributes to improve 105–107
  colors 311
  forms 183, 191–195
  images 139
  in JavaScript 684–687
  nav element for 89
  prefers-reduced-motion feature query 516
  refactoring for 684–689
  search element for 90
  SVGs 819
  tables 159–161
  text tracks on videos 214
  title element for 61
Accessible Fluid Type Generator 540
Accessible Rich Internet Applications (ARIA) 44, 105, 684
accordion sub-navigation 541
action attribute, form element 170–171
:active selector 336
adaptive design 506. *See also* responsive web design
adaptive icons 836
addEventListener() method 681, 682
addition (+) operator 613
addition assignment operator (+=) 616
address element 90, 114
Adobe
  Creative Cloud 18
  Dreamweaver 17
  Fonts 254
  Illustrator 789, 791, 820, 824
  Photoshop 18, 737, 740, 777, 789–791, 793
AdobeRGB 295
a element 115–116
  pseudo-classes for link feedback 337
  specific point link 126
Affinity Designer 740, 789, 791, 820
Affinity Photo 740, 790
::after pseudo-element 349–350
Aguilar, Justin 575
AI (artificial intelligence)
  image generator as image source 729
  upscaling with 791–794
AI Image Enlarger 794
align-content property 439–440, 490
align-items property 438–439, 493
align-self property 439, 491–492
Alliance for Open Media 748
alpha channel 305, 736
alpha transparency 756, 757
  AVIF 748
  CSS color values 305
  PNG-24 736
  WebP 743
alt attribute
  img element 65, 136, 138–139
  picture element 148
alternative text (alt text) 138
alt value as null, for decorative images 139
ampersand (&)
  character references 107
  nested selectors 597
ancestors, document tree 234
anchor element. *See* a element
and operator, logical 592
Andrew, Rachel 501
Android Studio 546
angle brackets (< >) 58
angle, transforming 562–563
AngryTools CSS Animation Kit 575
Angular 704, 708
Animate.css tool 575
animation
  APNG 739
  CSS properties 573, 578
  CSS resources 572
  GIF format and 746
  inspectors 574
  keyframe animation 571–575
  SVG images and 816
  tools for 572, 575, 817
  WebP image 743
animation-direction property 573
animation-fill-mode property 573
animation frames 747, 748
animation-iteration-count property 573
animation-play-state property 573
anonymous function 681
anonymous text 76
anti-aliasing 757
:any-link selector 334
Apache server software 24, 170
API (application programming interface) 209
APNG (Animated PNG) format 739
appearance property 581
appendChild() method 667, 668, 669, 671
Arango, Jorge 13
arbitrary substring attribute value selector (*) 332
Archibald, Jake 825
area names, positioning items using 482
argument 636, 639
ARIA (Accessible Rich Internet Applications) 44, 105, 684
aria-label attribute 685
aria-pressed attribute 684
arrays, JavaScript 652–653
arrow functions 707
article element 88, 114
artifacts in JPEGs 734
artificial intelligence. *See* AI
ASCII character set 59
Asian glyphs 268
aside element 89
aspect-ratio property 544
ASP.NET 171
assignment operator (=) 611, 615
asynchronous code 706
atomic design concept 531
Atomic Design (Frost) 531
attribute minimization 801
attribute nodes 674–677
attribute properties, JavaScript 676
attributes 65–66
  ARIA 106–107
  Boolean 66
  case sensitivity 331
  escaped characters in values 108
  global 105, 857–859
  id and class 103–105
  input elements (list) 199
  SVG image 144, 815
  XML requirements for 801
attribute selectors 330–332
Audacity 210
Audio Converter 210
audio element 209, 210, 213
audio files, encoding tools 210
auditory impairment 43
Auswöger, Martin 787
authoring 10
author style sheet 236
autocomplete options, forms 179
auto-fill value, Grid Layout 496–497
auto-fit value, Grid Layout 496–497
automating image production 794–796
autoplay attribute, video element 212
Autoprefixer 589
auto value, overflow property 366
AV1 Image File (AVIF) 747–748
AV1 video codec 209
Awwwards 549
axis, Flexbox 428

B

backend development 10, 12, 171, 711
background-attachment property 314
background-blend-mode property 314
background-clip property 308, 312
background-color property 307–309, 312
background-image property 311–313
background images 311–320
  adding 311
  adding multiple 316–317
  properties for controlling 313–315
  property list 328
  shorthand background property 317–318
background-origin property 314
background-position property 314, 318
background property 317–318
background-repeat property 314
background-size property 314, 319–320
backslash (\)
  avoiding in URL pathnames 118
  versus slash 58
backticks (`), string template literals 618
banner role, header element 106
Barker, Michelle 596
Barvian, Maxwell 540
Baseline 588
baseline, text 275
::before pseudo-element 349–350
beginning substring attribute value selector (^) 331
b element 93, 114
Bellamy-Royds, Amelia 837
Berners-Lee, Tim 24
Best Website Gallery 549
Bezier curve 554
binary (index) transparency 738, 746, 756
BitBar 547
bit depth 777
bitmap image formats 134, 730, 772
blinking text (deprecated) 279
Blink rendering engine 26
block axis 462
block-axis alignment properties 490
block comments 609
block display type 76, 388
block elements 63
  browser treatment of missing tags in 69
  floating 399, 400
  maximum and minimum dimensions of 365
  sizing 360–366
  specifying dimensions 360–365
block formatting context 406
block-level grids 462
blockquote element 82, 114
block scope 640
block-size property 361
blur event 680
blurring images 776
body element 59, 74, 87, 308
bold text
  b element 93
  font-weight property 264, 265
  strong element 92
Boolean attribute 66
Boolean values, JavaScript 615, 646
Bootstrap 600
border-block-color property 374
border-block-end-color property 374
border-block-end property 375
border-block-end-style property 371
border-block-end-width property 373
border-block-start-color property 374
border-block-start property 375
border-block-start-style property 371
border-block-start-width property 373
border-block-width property 373
border-bottom-color property 374
border-bottom-left-radius property 377
border-bottom property 375
border-bottom-right-radius property 377
border-bottom-style property 371
border-bottom width property 373
border-box method 363
border-color property 374
border-end-end-radius property 377
border-end-start-radius property 377
border-image-outset property 380
border-image property 380
border-image-repeat property 380
border-image-slice property 380
border-image-source property 380
border-image-width property 380
border-inline-color property 374
border-inline-end-color property 374
border-inline-end property 375
border-inline-end-style property 371
border-inline-end-width property 373
border-inline-start-color property 374
border-inline-start property 375
border-inline-start-style property 371
border-inline-start-width property 373
border-inline-width property 373
border-left-color property 374
border-left property 375
border-left-style ptoperty 371
border-left-width property 373
border property 375
border-radius property 377–378
border-right-color property 374
border-right property 375
border-right-style property 371
border-right-width property 373
borders 371–379
  border colors 374–375
  border images 380
  border style 371–373
  border width 373
  combining style, width, and color 375
  in element box 360
  elliptical corners 377
  rounded corners 377–378
border-start-end-radius property 377
border-start-start-radius property 377
border-style property 371–372
border-top-color property 374
border-top-left-radius property 377
border-top property 375
border-top-right-radius property 377
border-top-style property 371
border-top-width property 373
border-width properties 373
border-width property 373
bots 129
Box Alignment Module 436
box drop shadows 389
box model 239, 359–394
  assigning display types 387–388
  borders 371–379
  element box 359
  margins 381–385
  padding 366–369
  properties (list) 392–393
  specifying box dimensions 360–365
box-shadow property 389
box-sizing property 361, 363
Boxy 821
bracket notation 653
breakpoints, choosing 529–530
breaks, adding in lines and words 99–100
br (line break) element 64, 99, 114
Brotli 825
browser prefixes 589
browsers 25–26
  challenges of designing for different 26, 37

default margins 381
developer tools 244–246
history 24
HTTP status responses 34
ignored information in HTML 57
image downloading 140
popular 19
progressive enhancement to deal with different 40–41
rendering engines 26
SVGs 802, 831
BrowserStack 547
browser support
AVIF 748
clamp() function 540
clearfix hack 406
container queries 526
CSS feature detection 587–590
CSS variables 596
Flexbox 428
Grid Layout 460
JPEG-XL 749
location pseudo-classes 334
strategies for 590
WebP images 744
Browsersync 546
bulleted lists, changing markers 284–287
button element 180
buttons
checkbox buttons 184
custom buttons 180
images as 180
inputs for web forms 180
radio buttons 182, 581
submit and reset 180
bye() function 681

C

cache and caching
image loading advantages 137
SVG disadvantage for 145
Cagle, Kurt 837
Cailliau, Robert 24
calc() function 539
calling JavaScript functions 637
camel case 611
Can I Use, online resource 27, 588
canvas, adding background color 307
Canvas API 217, 218–219
canvas element 217–219, 816
capitalization
changing text 279
of element names 58
fonts 251
caption element 160, 163
Carnes, Beau 284
cascade, style 235–239
Cascading Style Sheets. *See* CSS
case-insensitivity, elements 58
Catlin, Hampton 593
CDNs (content delivery networks) 794
cellpadding attribute (deprecated) 159
cellspacing attribute (deprecated) 159
cells, table structure 156–157
centering
with Flexbox 442
with Grid Layout 489–492
with margins 382
text lines 277
character encoding 59
character entity references 107, 108, 109
characters, escape 107–110
character spaces, browser handling of multiple 57
@charset rule 353
ChatGPT 730
checkbox buttons, forms 184, 581
checked attribute 66, 183, 184
child-indexed selectors 340–344
child selector (>) 271, 330
Chinnathambi, Kirupa 572
chroma values, OKCH color model 303
Chrome
accessing device toolbar 518
animation inspector in 574
Canary release 587
CSS Shapes Editor 409
developer tools 629
Grid Inspector 477, 478
JPEG encoding library 749
Mobile FIRST plug-in 547
Responsive Image Linter 787
turning on feature flags in 587
chrome, browser UI 244
ch unit 537
CIELAB color space 304
circle element, SVG 144, 802
circle() notation, text wrapping 408
cite attribute, q element 94
cite element 95
clamp() function 539, 540
Clark, John 543
class attribute
HTML elements 104–105
JavaScript functions for handling 677–678
classList.add() method 677
classList.contains() method 678
classList.remove() method 678
classList.toggle() method 678
class selector (.) 271, 272, 330
"Clearfix" hack 406
clearing floated elements 401
clear property 402
click event 680
client software 25, 711
clip art, as image source 728–729
Clipart.com 728
clipPath element, SVG 802
clipping, SVG 806
closing tag 58
cloud font embedding services 254
Cloudinary 788, 795
CMSs (content management systems) 35, 171, 781, 796
CMYK color model 294
codec, video and audio 209
coded character set 59
code editors 16–17
CodeKit 17
CodePen 17, 217, 575
code points, in character sets 59
Codrops blog 575
coercion, JavaScript 620
cognitive impairment 43
col element 163
colgroup element 163
collaborative skills 15
collapsing margins 382–383
color() function 299, 302, 304
color map 737
color media features 515
color-mix() function 304
color model 293
color pickers 191, 298–299, 302
@color-profile rule 304
color property 306
colors 293–311
accessibility 311
background color property 307–309
for borders 374–375
Color Module Level 4 304
foreground color property 306
gradients 321–326
opacity of 309
property list 328
values for 296–306
HSL color 301–302
names 296–297
OKLCH color 302–303
RGB color 298–301

transparency 305
color spaces 293–295
color stops, gradients 321
color table 737
cols attribute, textarea 176
colspan attribute, table 158
column combinators (||) 271
column-gap property 493–494
column group elements 163
columns and rows, HTML tables 156
column spans 158
combinators (selectors) 270
command-line tools 21, 740
comments
HTML 57, 63
JavaScript 609
style sheet 230
CommonJS 696
communication skills 15
comparison operators 614
compiler 710
The Component Gallery 534
components
building JavaScript frameworks with 701–702
component-based design 531
design system library of 533
Compressor 781
comps 8
concatenating strings 619–620
condensed font 266
conditional loading 534
conditional statements 614, 645–652
defensive programming 650
if...else statement 646
logical operators 648
testing for opposite of statements 649
conditional styles, applying with @container 525
conic gradients 321
console, JavaScript 628–631
constant width typefaces 253
const keyword 624, 636
@container 525–526
container format, media 209
container-name property 525
container queries 512, 523–528
applying conditional styles with @container 525–526
browser support and fallbacks 526
identifying containers 523–524
container query units 241, 526
container-type property 524, 525
containing blocks, positioning 415–416
containing context 524
content area, in element box 360
content-based breakpoints 531
content-based sizing, Grid Layout 473
content box, sizing 362
content delivery networks (CDNs) 794
content hierarchy 531–534
content management systems (CMSs) 35, 171, 781, 796
content parity 534
content property, generated content 350
content strategists 13, 14
contextual selectors 268–270
contrast issue for colors, accessibility 311
Convertio 748
Cope, Rachel 572
copyright issue, images 137
Corel Draw 19, 820
Corel PaintShop Pro 19
Coyier, Chris 273, 323, 364, 454
createElement() method 667, 671
createTextNode() method 673
Creative Cloud 18
Creative Commons 727–728
critical thinking skills 15
cross axis, Flexbox 431, 438–439
crossorigin attribute, video element 212
cross size 432
CSS2.1 specification 251, 278, 284, 285, 514
CSS Animation Cheat Sheet 575
CSS animation resources 572
The CSS Animations Level 2 spec 572
CSS (Cascading Style Sheets) 10, 33, 225–248, 579
adding to HTML document 68–69
alternative to an image 770
animation 571–575
background properties (list) 328
benefits 225–226
box model. *See* box model
box properties (list) 392–393
browser developer tools for 244–246
colors and backgrounds (list) 328
CSS layers 598
declaration block 228
external style sheets 350, 352–354
feature detection 587–590
Flexbox properties (list) 457
floating and positioning properties (list) 425
font and text properties (list) 290
form styling 197
frameworks 600
Grid Layout properties (list) 504
handling conflicting styles (cascade) 235–239
history 240
inheritance of styles 232–235
keyframe animation properties (list) 578
list styling 80, 284–287
math functions 539
media queries within 517
nesting 596–597
normalizers 585
preprocessors 593
progressive enhancement 40–41
resets 584
responsive SVGs as background images 832
selectors 329–358, 861–864
sprites 811
step-by-step workings of 226–232
styling SVGs 144, 813–815
styling tips for forms 579–582
table styling 156
text formatting. *See* formatting text with CSS
text wrapping with CSS shapes 407–408
transform properties (list) 578
transition properties (list) 578
units of measurement 241–244
variables 592–595
CSS Color Modules 295, 301, 302, 304
CSS Containment Module 527
CSS custom properties 592, 594–596
CSS Design Awards 247
CSS Flexbox Layout Guide (Coyier) 454
CSS Flexible Box Layout Module. *See* Flexbox
CSS Fonts 257
CSS Grid Layout Module. *See* Grid Layout
CSS Layers 598
CSS Multi-Column Layout Module 455
CSS Nectar 247
CSS Nesting 596–598
CSS reset 584–585
CSS Shape Path Editor 409
CSS Shapes Editor 409
CSS Text Decoration Module 282
CSS Text Modules 282
CSS-Tricks 129, 247, 837
CSS variables 592, 594–596
CSS-wide keywords 250
CSS Winner 549

Cubic-Bezier.com tool 555
cubic-bezier keyword, transitions 554
curly braces ({ })
  declaration block 228
  nesting consideration 513
cursive typefaces 253
custom elements 699
custom form widgets 194
Cygwin emulator 21

D

D3.js 697, 818
Dall·E 730
Dan's Tools 257
data attribute, object element 146, 208
data element 98, 114
datalist element, forms 179
data visualization, and SVGs 818
date selection, forms 188–189
datetime attribute, time element 97
dd element 81
declaration 229
declarations, JavaScript functions 636
decorator, in Angular 708
decrementing (--) operator 616
deep linking, video or audio track 214
default attribute, track element 216
default files 28–29
default rendering of elements 64
defensive programming, JavaScript 650
deg unit 562, 567
del element 98
Deno 713
deno command 713
density (x) descriptor 786
desc element, SVG 819
descendants, document tree 233–235
descendant selectors 268, 271
descendent selectors 329
description lists 81–82
designers, as image sources 730
design systems 9, 532
  component library 533
  design tokens 533
  documentation in 534
  pattern library 533
  style guide in 533
  UI tool library 533
Design.Systems 534
design tokens 533
desktop browsers 25
details element 197
developer tools 244–246, 629
device-CMYK() function 304
device-independent pixels (dp) 755
device lab 546
device pixel ratio (DPR) 754
devices 38–39, 546. *See also* mobile devices
device width 507
DevTools 244, 245
dfn element 95
dialog element 197
dir attribute, html element 100
directories
  images 137
  names in URLs 28
  and relative links 118–124
disabilities, designing to accommodate 43
disabled attribute, forms 177
disclosure widget, forms 197
Discord 721
Display P3 295, 302, 304
display property 285, 387–388, 590
displays (screens)
  high-density 754, 787
  quality media features 515
disposal method, frame animation 747
dithering 779
div element 102
division assignment operator (/=) 616
division (/) operator 614
dl element 81–82
DNS (Domain Name System) 25
DNS servers 25
DOCTYPE declaration 59
document object 667
Document Object Model (DOM) 12, 61, 666–678
  adding text 671
  attribute nodes 674–677
  document object 667
  element nodes 667–669
  text nodes 673
document outlines 76
document type declarations. *See* DOCTYPE declarations
document type (DOCTYPE) declaration 59
domain names 27
Domain Name System (DNS) 25
Don't Repeat Yourself (DRY) coding 635, 810
"dot-dot-slash" (../) directory 122
dot notation 610, 611
dots per inch (dpi) 753
double colon (::), pseudo-element selectors 348
double curly braces ({{ }}), template properties 709
double equals operator (==), equivalency 615
double quotes (")
  strings 618, 619
  wrapping attribute values in 66
double slashes (//), comments 609
dp (device-independent pixels) 755
dpi (dots per inch) 753
DPR (device pixel ratio) 754
Drasner, Sara 816
Dreamweaver 17
drop-down menus, forms 185, 581
drop shadows
  element boxes 389–390
  text shadow 281–282
Drupal 781, 796
DRY (Don't Repeat Yourself) coding 635, 810
dt element 81
Dunbar, Natalie Marie 14, 535
Dunham, Ethan 254
dynamic versus static websites 34

E

eager value, loading attribute 141
ease-in keyword, transitions 554
ease-in-out keyword, transitions 554
ease keyword, transitions 554
ease-out keyword, transitions 554
easing function 554
Eckles, Stephanie 527
ECMAScript 607
ECMAScript Modules (ESM) 696
Eden, Daniel 575
8-bit indexed color format 736–738, 746
Eisenberg, J. David 837
element box 359
element display state pseudo-class selectors 347
element inspector, developer tools 245
element nodes 666, 667–669
elements 57–61
  attributes. *See* attributes
  default rendering of 64
  and document tree structure 233–235
  embedded media (list) 221–222
  forms (list) 199
  identifying 61–64
  images (list) 151

links (list) 132
nesting 80, 95
non-replaced 136
replaced 136
resources for 72
SVG 141
syntax 57
tables (list) 166
text (list) 114
void (empty) 32, 64
element selectors 228, 268, 272, 329
ellipse element, SVG 802
ellipse() notation, text wrapping 408
elliptical corners 377
email addresses, linking to 129
email input type, forms 178
embedded content 205–222
bitmap images, SVG 806
elements list 221–222
iframe element 205–207
object element 208
streaming media 211
video and audio 208–216
web fonts 251, 253–254
embedded JavaScript 627
embedded style sheets 230
embed element (outdated) 146, 208
em element 63, 234
empty elements 64
:empty selector 339
emulator 546
em unit 242–243, 259, 260–261
encoding, form data 168
encoding method (codec), video and audio 209, 210
encoding type (enctype), file selection input 188
encryption of data, and HTTPS 27
ending substring attribute value selector ($) 331
end state, for transitions 552
end tag 58
EnvatoTuts+ 217
EPUB 801
equals sign (=)
assignment operator, in JavaScript 622
attribute values 66
equal to (===) operator 614
equipment, web development 15–16
error messages, web page 34
escaped characters 107–110, 618
ES Module specification 699
event delegation 682
event handler 680
event handling (JavaScript) 680–682
event listeners 681
event propagation 682
events, JavaScript 678, 679–683
exact attribute value selector 331
exercises
absolute positioning 420
accessibility, JavaScript 686–688
adding a style sheet 68
adding attributes, JavaScript 679
adding button element with JavaScript 675
adding images 66–67
adding SVG to page 147
adjusting flex and order 454
alignment, Grid Layout 495
animation, SVG 818
attribute selectors 333
background images 316–317
borders 379
child-indexed selectors 342
complex table structure 164–165
conditional statements 617
container queries 527–528
content without markup 55–56
event listeners added to buttons 683
external JavaScript 628
external style sheet 354
fixed positioning 421
flexbox alignment properties 443–444
floating images 403
font family settings 258
font size setting 262
font-weight property 265
formatting text 282
forms 181–182, 184, 187, 195
functional programming 638
glow filter 811
goals and assets assessment 22
grid layout 498–499
identifying text-level elements 101
iframe 207
implicit grids 467–468
inline style application 232
introduction to CSS 231
introduction to JavaScript statements 612
introduction to text markup 62
JavaScript challenge 689
JavaScript console 631
JavaScript numbers, strings, and variables 625
JPEG and PNG format comparison 741–742
linking images 142
linking to document fragments 128
linking to external sites 117
linking to files within directories 120
linking to higher directory 123
linking to same directory 119
link-related pseudo-selectors 338
lists, using DOM 659
loops, JavaScript 657–658
margin space around elements 386
marking up a recipe 85
minimal markup 60
navigation bars with Flexbox 435
News page markup 111–112
:not selector 346
number challenge, JavaScript 632
optimizing images 782–784
oval background images 808
padding 369
passing values 643
placing items on grids 485–486
pseudo-elements 351
responsive images 512
RWD example 519–521
scaling SVGs 834–835
selectors 275
setting up grids 475–476
shapes around floats 410
spanning table columns 158
spanning table rows 159
string combination 620
style sheet first look 228
SVG shapes 805
symbols in SVGs 812
testing for multiple conditions 650
transforms 569
transitions 559–560, 568
transparent images 758–761
typed structural pseudo-classes 344
video embedding 216
viewing a page source 32
WebP 745
writing relative URLs 125
expanded font 266
Experimental Layout Lab 501
explicit animation 571
explicit association, label element 193
explicit grids 469–476
Export As function (Photoshop) 759
export keyword, JavaScript modules 696
Extensible Markup Language (XML) 141, 801
eXtensible Stylesheet Language Transformations (XSLT) 819

external JavaScript 627
external links 117
external style sheets 230, 350, 352–354
  importing with SVGs 814
  using media queries within 517
extranet 25

F

fallbacks for nonsupporting browsers
  container queries 526
  object element 146
  picture element 148, 752
  rule order for 239
falsy 649
fantasy typefaces 253
favicon images 762–764, 765
Favicon.io 765, 766
feature flag 587
feature queries 590–592
FFmpeg 210
fieldset element, forms 193–195, 581
figcaption element 84, 114, 138, 160
Figma 18, 740, 743, 789, 791, 824
figure element 84, 114, 160
file compression tools, SVGs 825
file formats, MIME types 148
file input 187
file management tools 20–21
file selection input, forms 187
File Transfer Protocol (FTP) 20
file upload widget 187
filter effects, SVG 809
filter element, SVG 802
filter primitives, SVG 809
Firefox
  animation inspector in 574
  CSS Shape Path Editor 409
  developer tools 246, 629
  support for 588
  turning on feature flags in 587
firewalls 25
:first-child selector 340
::first-letter selector 348
::first-line selector 348
:first-of-type child selector 343
fixed positioning 411, 421
fixed-width layouts 509
Flanagan, David 608, 720
flex-basis value 448
Flexbox 427–458
  alignment techniques 434–439, 441–443
  behavior 430
  browser support for 428
  centering elements with 442
  controlling flow within flex containers 430–434
  CSS properties (list) 457
  nesting flexboxes 430
  for page layout 453–454
  reordering flex items 450–452
  resources 454
  setting up a flex container 428–430
flex, concept of 427, 445
flex container 428–434
flex-direction property 430–432
flex-flow property 434
flex-grow property 445–446
flexible images 507, 510–512
flexible layout 507
flex items 429
  aligning with margins 441–442
  behavior 430
  controlling alignment of 434–439
  expanding 446–448
  properties 446
  providing initial size 448–449
  reordering 450–452
flex property 445
flex-shrink property 445, 447–448
flex values 445–450
flex-wrap property 432–434
Flickr 728
float containment 404–406
floating 395, 396–410
  block elements 400
  clearing elements 401
  inline text elements 398
  key behaviors 398
  multiple elements 404
  properties list 425
float property 396
flow direction, Flexbox 430–432, 466
flow-root display 387
fluid layouts 509–510
Fluid Type Scale Calculator (Utopia) 540
fluid typography 538–539
focus event 680
:focus selector 335, 337
:focus-visible selector 334, 337
:focus-within selector 334
@font-face rule 251, 256
font-family property 251, 256
font-feature-settings property 268
font-kerning property 268
font-language-override property 268
font properties 249–268
  condensed and expanded 266
  name (typeface) 251–258
  shorthand font property 267
  size 259–263
  small caps 266
  style (italics) 265
  weight (boldness) 264, 265
font property 267
font-size-adjust property 268
font-size property 259–263
Fontspring 254
Font Squirrel 254
font stacks 251, 257
font-stretch property 266
font-style property 265
font-synthesis property 264
font-variant-alternate property 268
font-variant caps property 268
font-variant-east-asian property 268
font-variant-ligature property 268
font-variant-numeric property 268
font-variant-position property 268
font-variant property 266, 267
font versus typeface 251
font-weight property 264, 265
footer element 87, 114
for attribute, label element 193
forEach() loop 654
foreground color properties 306
for loop 654–655
formatting text with CSS 249–292
  capitalization 279
  letter and line spacing 280
  list marker changes 284–287
  properties (list) 290–291
  shadow offsets 281–282
  text line adjustments 275–278
  underlines 278
form controls 167, 173–191
  button element 180
  checkbox buttons 184
  color input type 191
  custom buttons 180
  date and time selection 188–189
  file selection 187
  hidden 188
  image buttons 180
  and labels 192
  menus 185–187
  numerical inputs 190
  radio buttons 182
  styling note 170
  submit and reset buttons 180
  text entry 175–179

form element 169–172
forms 167–204
  accessibility features 183, 191–195
  action attribute 170
  controls. *See* form controls
  elements (lists) 192, 199–201
  form element 169–172
  layout and design 196–198
  method attribute 171–172
  name attribute 172
  styling tips for 579–582
for. . .of loop 654
forward-compatibility 39
Foundation 600
fragment identifier 126
fragments, linking to document 126–128
Frain, Ben 548
frame delay, animation 747
frameworks
  CSS 600
  JavaScript 701–712
FreeCodeCamp 721
Freemake 210
Freepik 729
frontend development 10–12, 711
Frontend Dogma 549, 598
Frost, Brad 10, 38, 531, 537, 549
fr unit, Grid Layout 471–472, 490, 494
FTP (File Transfer Protocol) 20
full-stack developers 12
functional notation 299
function scope 640
functions (JavaScript) 610
  calling 637
  declaring 636
  naming 637
  passing values to 639
  returning values from 642
  variable scope 640–641

G

gamma correction 738
gamut, color 294
gap properties 493–494
gauge, meter element 192
Gecko rendering engine 26
g element, SVG 802
generalizing a function 651
generated content 349–350
generic font families 252–253
generic text elements (div, span) 101–105
getAttribute() method 676
GET method, form element 171, 172
Getty Images 727
GIF (Graphic Interchange Format) 746–747, 777–779
Gigapixel 794
GIMP 19, 740, 790
Git for version control 20
global attributes 105, 857–859
global scope 641
Global Stats (StatCounter) 531
glyphs, typography 268
Go (GoLang) 171
Google Fonts 254, 255, 729
Google Lighthouse 46
gradient line 321
gradient ray 323
gradients, color pattern 321–326
graphical browsers 25
graphical user interface (GUI) 740
graphic designers 8
Graphic Interchange Format (GIF) 746–747, 777–779
grayscale color space 293
greater than (>) operator 614
greater than or equal to (>=) operator 614
GreenSock Animation Platform (GSAP) 695, 817
grid-area property 484–485
grid areas 461, 482–483, 491–493
grid-auto-columns property 465
grid-auto-flow property 466
grid-auto-rows property 465
Grid by Example 501
grid cell 461
grid column 462
grid-column-end property 479–480
grid-column property 482
grid-column-start property 479–480
grid containers, aligning tracks within 462, 490–491
grid density 466
Grid Garden (game) 501
Grid Inspector (Chrome) 477, 478
grid items
  reordering 489
  rotating 562–564
  stacking 489
Grid Layout 459–504
  alignment and spacing 489–494
  block axis 462
  containers 462
  CSS properties (list) 504
  default grid behavior 463–468
  explicit grids 469–476
  grid property 487–489
  implicit grids 463–468
  inline axis 462
  items 462
  placing items on the grid 477–486
  resources 501
  responsive grids 494–500
  terminology 461–462
grid lines 461, 477–482
grid property 487–489
grid row 462
grid-row-end property 479
grid-row property 482
grid-row-start property 479–480
grid structural pseudo-class selectors 347
grid-template-areas property 483
grid-template-columns property 470–471
grid-template property 488
grid-template-row property 470–471
grid tracks 462, 469–471
Grigsby, Jason 794
grouped selectors 240, 268, 329
grouping elements 82, 90
grouping options in menus 186
Grunt 824
GSAP (GreenSock Animation Platform) 695, 817
GUI (graphical user interface) 740
Gustafson, Aaron 665, 721
Gzip 825

H

h1 (heading level 1) 76–77
h2 (heading level 2) 76–77
h3 (heading level 3) 76–77
h4 (heading level 4) 76–77
h5 (heading level 5) 76–77
h6 (heading level 6) 76–77
H.264 video codec 209
H.265 video codec 209
halo 756
HandBrake 210
hanging indent 277
hanging-punctuation property 282
:has() selector 346
HDR (high dynamic range) monitors 302, 743, 748, 749
head element 59, 74, 230
header element 86, 114
headers attribute, tables 161
headers, table 157, 160–161

heading elements (h1 to h6) 64, 76–77, 88
Head, Val 572
hearing impairment 43
height attribute
  canvas element 218
  iframe dimensions 206
  img element 139, 140
  video element 211
height property 361, 362
  border-box method 365
  and display property 362
  svg element 830
Herman, Nick 255
hexadecimal (hex) values, RGB color 300–301, 306
hidden form controls 188
hidden input type 188
hidden value, overflow property 366
hi() function 681
high-density displays 754, 787
high dynamic range (HDR) monitors 302, 743, 748, 749
High Efficiency Video Coding (HEVC) 209
HLS (HTTP Live Streaming) 211
Hoefler & Co. 254
Hoffman, Jay 590
horizontal rule (hr element) 78
horizontal text alignment 277–278
hostname, domain 28
hotlinking 137
:hover selector 335, 337
href attribute, a element 115, 116–117
hr element 78
HSB versus HSL color models 301
hsla() color function 305
hsl() color function 302
HSL color model 294, 301–302, 305
HTML5 30, 91
HTML5 for Publishers (Kleinfeld) 218
html element 59, 74
HTML (HyperText Markup Language) 10, 11, 30–32
  creating a simple page 51–74
  naming conventions for files 56
  progressive enhancement 40–41
  resources for 72
  troubleshooting 69–70
  validating documents 71–72
HTML Living Standard 72, 75
HTMLMediaElement API 213
HTML Template specification, web components 699
HTTP header 34, 59
HTTP (HyperText Transfer Protocol) 11, 24
HTTP Live Streaming (HLS) 211
HTTP requests, reducing for site performance 46
HTTPS (HyperText Transfer Protocol Secure) 27, 28, 171, 716
HTTP status responses 34
hue values
  HSL color model 301
  OKCH color model 303
hwb() color function 304
hypertext links 24. *See also* links
HyperText Markup Language. *See* HTML
HyperText Transfer Protocol. *See* HTTP
hyphen (-), custom element names 699
hyphen-separated attribute selector 332
hyphens property 282

I

i18n (internationalization) 100
IANA (Internet Assigned Numbers Authority) 59
icoconverter.com 763
ICO format 762
Icon Finder 729
icons 728–729, 764–765
id attribute 103
  forms 193
  as global attribute 105
  linking to fragments 126
ID selector (#) 229, 271, 330
i element 93, 114
if...else statement 646
if keyword 645
iframe element
  nested browsing context 205–207
  for SVG (outdated) 146
Illustrator 789, 791, 820, 824
image CDNs 795
image formats 730–752
  alternative formats with picture element 148–149
  AVIF 747–748
  bitmaps 134, 730, 772
  choosing 732, 750
  GIF 746–747
  JPEG 733–735
  JPEG-XL 749–750
  list 750
  naming and identifying 731
  PNG 735–740
  supported by web 134–135
  SVG. *See* SVG
  WebP 743–745
ImageKit 795
ImageMagick 796
ImageOptim 781
Image Optimization (Osmani) 725
image production 9, 769–798
  bitmap formats 772
  and CSS 770
  number of images 773
  optimizing images 773–784
  responsive images 772, 785–788
  SVG 771
  tools, tips, and techniques 789–793
images 133–152, 725–768, 769–798, 799–838
  accessibility 139
  adding to HTML document 64–67, 133
  automating production 794–796
  background images, CSS 311–320
  buttons based on 180
  creating 726
  cumulative quality loss 734
  editing tools 18–19, 740
  elements (list) 151
  fallbacks with picture element 752
  favicons and site icons 762–766
  flexible, RWD 510–512
  formats. *See* image formats
  gradients as 321
  HTML handling of 33
  link pathnames to 125
  links from 115
  optimizing 46. *See also* image production
  resolution and reference pixels 752–756
  in RWD 511, 543
  sources for 726–729
  transparency 756–761
  troubleshooting errors 70
image vectorization 789
img element 33, 64, 135–141
  as fallback for picture 148
  link attached to 115
  link pathnames to 125
  SVG 144, 819, 830, 831
imgix 795
ImgLarger's AI Image Enlarger 794
immutable variables 624
implicit animation 571
implicit association, label element 193
implicit grids 463–468, 489
!important style identification 236, 272

@import rule 230, 352, 354
import statement, web components 695
inclusive design 45
incrementing (++) operator 616
index, array 652
index (binary) transparency 738
indexed color 737
index file 29
information architecture 13
information designers 8
inheritance, CSS 232–235
inherit property value 250
initial containing block (html element), positioning 415
initializer, JavaScript variables 622
initial property value 250
Inkscape 821, 824
inline-axis alignment properties 490
inline axis, Grid Layout 462
inline display type 388
inline elements 63, 90–101
  floating 398
  margins on 384
inline-end logical value 396
inline-flex value 429
inline frame (iframe) 205
inline grids 462
inline-size property 360
inline-size value 525
inline-start logical value 396
inline styles 232, 272, 814
inner display behavior 388
inner edges, element box 360
innerText property 611, 669, 671, 672
input control pseudo-class selectors 347
input element, forms 90, 174–176, 180, 182–184
input types, design considerations 196
input value pseudo-class selectors 347
ins element 98
insertBefore() method 667, 668, 669, 673
interaction design (IxD) 5–7
interlacing 746
internal style sheets, SVG 814
internationalization (i18n) 100
Internet Assigned Numbers Authority (IANA) 59
Internet Explorer 20, 587
internet versus web 23–24
Interop Project 588
intranets 25
intrinsic design 509
IP addresses 25
:is selector 345
iStockPhoto 727
italic text
  em element 92
  font-style property 265
  i element 93
IxD (interaction design) 5–7

J

Java 170, 607
JavaScript 33, 607
  accessibility 684–689
  for animation of SVGs 817
  arrays 652–653
  beyond the browser 711–714
  Canvas API 217
  commenting code 609
  conditional statements 645–652
  console 628–631
  customizing input elements for 180
  custom video players 213
  DOM scripting 11–12. *See also* Document Object Model
  dot notation 611
  ECMAScript 607
  embedded versus external 627
  events 678, 679–683
  form design capability 168
  functions 636–645
  generating new markup 665–678
  interactivity of SVGs with 815–817
  introduction to scripts 608–612
  lack of application to SVG images 144
  learning 608
  libraries 694–698
  loops 654–657
  minimizing for site performance 46
  modules 696
  Node.js 170
  numbers and math 613–617
  objects 611
  packages, npm 713
  progressive enhancement 40–41, 664–665
  statements 612
  strings 618–620
  SVG access and manipulation 145
  variables 621–625
  web components 698–700
JavaScript class 699
JavaScript engine 712
JavaScript frameworks 701–711
  building with components 701–702
  popular 705–710
  pros and cons of 704
  using 702
JavaScript Object Notation (JSON) 718, 765
JavaScript runtime 712
Joint Photographic Experts Group (JPEG) 733–734, 735, 775–776
Jorgensen, Erica 535
JPEG image format 148
jpegli 749
JPEG-XL 749–750
JPEG-XL Community 749
jQuery 694, 695
jsDelivr 695
JSON (JavaScript Object Notation) 718, 765
justify-content property, Flexbox 436, 445, 490
justify-items property, Grid Layout 493
justify-self property, Grid Layout 491–492

K

kbd element 96
kch() color function 304
Keith, Jeremy 719
kerning, font 268
keyframe animations 571–575
  establishing keyframes 571
  inspectors 574
  properties 573–574
  resources 572
  review list 578
  tools for 575
  when to use 574
@keyframes rule 571
key/value pairs, web storage 689
kind attribute, track element 215
Kleinfeld, Sanders 218
Kraken 795

L

lab() color function 304
label attribute
  grouping menu options 186
  optgroup element 186
  track element 216
label element 192
labels, form 192, 195, 196
landmarks, ARIA roles as 106
lang attribute, html element 59, 100
language tag 59

:last-child selector 340
:last-of-type child selector 343
Latin-1 character set 59
Latin, Matej 284
@layer rule 598
layout engines, browser 26
layout patterns, RWD 535
lazy loading of images 140–141, 775
lazy value, loading attribute 141
leading 242. *See also* line-height property
Leaner Style Sheets (Less) 593
legend element 193, 581
length property 653, 656
length units, CSS 259, 276
Lesiński, Kornel 757
Less (Leaner Style Sheets) 589, 593
less than (<) operator 614
less than or equal to (<=) operator 614
let keyword, JavaScript 621, 624
Let's Encrypt project 27
LetsEnhance 794
letter-spacing property 280
libraries, JavaScript 694–698
licensing of fonts 253
li element 117
ligatures 268
Lighthouse (Google) 46
lightness or luminosity, HSL color model 301
Lightning CSS 354
Lightning Design System (Salesforce) 533
Lindley, Cody 720
line adjustments, text 275–278
linear gradients 321–323
linear keyword, transitions 554
line-box 275
line break (br) element 64, 99
line-break property 282
line breaks, browser ignoring of 57
line comments, JavaScript 609
line element, SVG 802
line height 275–276, 537
line-height property 275–276
line length 518, 537
line names, assigning 478
line-padding property 282
linguistic pseudo-class selectors 347
link element 230, 353, 763
links 115–132. *See also* URLs
  element list 132
  to email addresses 129
  external 117, 127
  images as 138, 142
  internal 118–129
    to fragments 126–128
    to a higher directory 122–123
    to images 125
    to a lower directory 120–121
    to new browser window 128–129
    site root relative 124
    to a subdirectory 121
    within a directory 119
  pseudo-classes 334–337
  security 335
  styling 334
  telephone 130
  usability tips 334
:link selector 334
Linux environment 16
list attribute, forms 179
list elements (ul, ol, li, dl, dt) 78–82
list markers, changing in CSS 284–287
list-style-image property 287
list-style-position property 286
list-style property 287
list-style-type property 284–285
loading attribute, img element 140
:local-link selector 334
localStorage object 689
location pseudo-class selectors 334
logical AND operator (&&) 648
logical NOT operator (!) 649
logical operators 648
logical OR operator (||) 648
logical pixel 754
logical properties 361, 368
logical pseudo-class selectors 345–346
logic gates, JavaScript 646
log() method 630
loop attribute, video element 212
loops 654–658
loosely typed, JavaScript 703
lossless compression 736, 743, 746, 748
lossy compression 733, 743, 748
luminance (brightness) 807
Luxon 697

M

Macintosh environment 16
  running Windows in 16
  server software in 24
  text editor 54
main axis, Flexbox 431, 432, 436–438
main element 86, 114
MainNavigation component 701
main size 432
Mall, Dan 534
Marcotte, Ethan 41, 507, 531, 549
margin-block-end property 381
margin-block property 381
margin-block-start property 381
margin-bottom property 381
margin-inline-end property 381
margin-inline property 381
margin-inline-start property 381
margin-left property 381
margin property 276, 381, 416
margin-right property 381
margins 381
  adding to grid items 493
  aligning flex items with 441–442
  behavior of 382–384
  browser default 381
  centering with 382
  collapsing 382–383
  in element box 360
  on floated elements 398, 399
  on inline elements 384
  negative 384
  and typography 537
margin-top property 381
mark element 96
::marker selector 287
markers, list 284–287
markup language 10, 58
mask element, SVG 802
masking, SVG 807
MathML 801
math operators 613
matte feature 756, 761
max attribute, numerical form controls 190
max-block-size property 365
max-content property 474
max() function 539
max-height property 365
max-inline-size property 365
maxlength attribute, form input 176, 177
max-width property 365
MDN Web Docs 72, 246, 455, 501, 608
M-dot sites 42
measurement values, CSS properties 259
media 514–515. *See also* embedded content
  audio 209, 210, 213
  images. *See* images
  video 209, 210, 211–213
media attribute 353
MediaHuman Audio Converter 210

media playback state pseudo-class selectors 347
media queries 498, 512, 513–522
  for accessibility 516
  external style sheet control 353
  how to use 517
  media features 514–515
  media types 514
  as RWD component 507
  testing mobile devices for hover 336
Meiert, Jens Oliver 549
menus
  forms 185–187
  site navigation 541
metadata
  image optimization 780, 824
  in images 735, 743, 748
  media container 209
  meta element 59
  web app manifest 716
meta element 59, 74
meta tags 716
meter element, forms 192
method attribute, form element 171–172
methods, JavaScript 611, 638
Meyer, Eric A. 243, 273, 323, 501
Microsoft ASP.NET 171
Microsoft Edge 20
  animation inspector in 574
  Canary release 587
  developer tools 246, 629
  turning on feature flags in 587
MIME (Multipurpose Internet Mail Extension) types 135, 148
min attribute, numerical form controls 190
min-block-size property 365
min-content value 474
min() function 539
min-height property 365
minimized attributes 183
min-inline-size property 365
minlength attribute, form input 176, 177
minmax() function 472, 496
Min-Max-Value Interpolation tool (9Elements) 540
min-width property 365
mobile browsers 25
mobile devices. *See also* responsive web design
  complications of designing for 38
  hover limitation 336
  testing on 16
Mobile FIRST plug-in 547
Mobile First (Wroblewski) 196, 535
mobility impairment 43
mockups 8
modern-normalize (Sorhus) 586
modular style sheets 354
modules, JavaScript 696
monospace fonts 83, 96, 253
Morville, Peter 13
MP3 (MPEG-1 Audio Layer 3) format 210
MP4 (MPEG-4) audio format 210
MP4 (MPEG-4) video format 209
MPEG-DASH 211
Multi-Column Layout Module 455
multikeyword display values 388
multiline strings 618
multiline text entry field, form 176
multimedia producers 14
multiple attribute, forms 186, 188
multiplication assignment operator (*=) 616
multiplication (*) operator 614
mustache syntax 709
mutable variables 624
muted attribute, video element 212
MyComponent class 709

N

naked text 76
name attribute, forms 172, 176, 183, 184
named entity, character entity reference 107
named export, JavaScript modules 696
named functions 681
names of fonts, specifying 251–258
namespacing 641, 642
naming conventions
  form variables 173
  JavaScript functions 637
  JavaScript variables 623
Nanwani, Rahul 795
nav element 89
navigation, for RWD 541–542
negation selector, :not() 345
negative margins 384
negative values
  margins 384–385
  position 412
nested browsing context (iframe) 206
nested flexbox 430
nesting
  avoiding form element 169
  elements 87, 95, 102
  lists 80
  selectors 597
network speed, as design factor 38, 45–47
Network tool, browser 46
Neumann, Peter 787
new browser window, internal links to 128–129
newline character 612
next-sibling selector (+) 271, 330
nginx 24
nightly builds 587
node command 713
Node.js 170, 712
Node Package Manager (npm) 713
nodes, DOM 666
nodeValue property 674
non-breaking space ( ) 109
none display type 388
non-replaced element 136, 362
normal flow 387, 395, 401
normalizers 585
Notepad 53
not equal to (!=) 614
not operator (!) 592
:not() selector 345
The Noun Project 729
npm (Node Package Manager) 713
:nth-child() selector 340–341
:nth-last-child() selector 340–341
:nth-last-of-type() child selector 343
:nth-of-type() child selector 343
Nu Html Checker 71
null value, alt attribute 138
number character styles 268
numbered lists 80, 284
number input type, forms 190
numeric entity, character entity reference 107

O

object element 146, 208
object-fit property 318, 319
object-position property 318, 319
objects
  document 667
  JavaScript 611, 699
oblique value, font-style property 265
octothorpe symbol (#)
  for hexadecimal color values 300
  ID selectors 229, 271
offset properties 412
oklab() color function 304

oklch() color function 304
OKLCH color model 294, 302–303
ol element 80
onclick property 680
1-bit transparency 738
:only-child selector 340
:only-of-type child selector 343
opacity property 309
opening tag 58
open source fonts 254
open source software 24
Opera developer tools 246, 629
operators 611
  comparison 614
  math 613
  @supports rule 591
optimization
  image 773–784, 824
  SEO 14, 775
  site 46
Optimizilla 781
option elements, menus 185
Opus audio codec 210
Oracle VirtualBox 16
ordered lists 80, 285
order property 450–452
ordinal group, Flexbox 452
origin image, tiling backgrounds 314
or operator 592
Osmani, Addy 720, 725
outer display behavior 388
outer edge, in element box 360
outline-color property 376
outline-offset property 376
outline property 376
outlines, CSS 376
outline-style property 376
outline-width property 376
output element 192
overflow property 365
overflow-wrap property 282
overlay toggle 543
overlining text 278

P

padding
  box model 366–369
  in element box 360
  and RWD 537
padding-block-end property 366
padding-block-start property 366
padding-bottom property 367
padding hack, SVG 832
padding-inline-end property 366
padding-inline-start property 366
padding-left property 367
padding property 276, 367
padding-right property 367
padding-top property 367
pages. *See* web pages
PaintShop Pro 19, 740
palette 737
Parallels software 16
param element (outdated) 208
parameters, JavaScript functions 636
parentheses (), in JavaScript 681
parentNode property 669
parents, document tree 234
Park, Thomas 501
partial attribute value selector 331
password entry field, forms 177
path animators 817
path element, SVG 802
pathnames 118
pattern library 533
p element 75
perceived lightness values, OKCH color model 303
percentages
  in CSS 241
  font size 259, 262
  specifying position using 419
  text-indent property 276
Perfecto 547
performance 45–47
  cache and caching 137, 145
  external style sheet contribution to 350
  gradient impact on 322
  modular style sheet issue 354
  reducing HTTP requests 46
  SVG disadvantage for 145
period symbol (.)
  class selectors 272
  dot notation 611
persona 6
Peterson, Clarissa 518
Phoenix Code 17
phone links 130
photographs, as image sources 726
Photopea 19
Photoshop 18, 737, 740, 777, 789–791, 793
PHP scripting language 170
phrasing content 76
phrasing (inline) elements 63, 135
picture element 148–149
pixel density 753, 755
Pixelmator Pro 19, 740
pixels per inch (ppi) 752, 753
Pixlr 19
placeholder attribute, forms 176, 177
playsinline attribute, video element 212
plug-ins, browser 208
plus (+)
  addition operator 613
  next-sibling selector 271
  :nth-child selector 341
PNG-8 777–779
PNG-24 735–736, 776
PNG (Portable Network Graphic) 148, 735–741, 763, 776–778
pngquant 757
Polaris Design System (Shopify) 533
polyfill 590
polygon element, SVG 802
polygon() notation, text wrapping 408
polyline element, SVG 802
Polypane 819
Portable Network Graphic (PNG) 148, 735–741, 763, 776–778
positioning 395, 411–423
  properties for 425, 479–481
  specifying position 412, 417–418
  types of 411–412
positioning context 415
position property 411
PostCSS 354
poster attribute, video element 212
POST method, form element 171
posture, fonts 265
ppi (pixels per inch) 752, 753
pre element 83
pre-export best practices, SVGs 822–823
preload attribute, video element 212
preloader, browser 140
preloading fonts 257
preprocessors 593
presentational attributes 813
presentation layer 33, 227. *See also* CSS
preserveAspectRatio attribute, SVG 829
primary light colors (RGB) 295
priority, style rule 236
product managers 14
progress element 192
progressive enhancement 40, 664
progressive JPEGs 734, 776
progressive PNGs 739
progressive web apps (PWAs) 714–720, 762
project managers 14
promises, JavaScript 706
properties, CSS 229

adding multiple 232
custom (CSS variables) 595
inheritance of 232–235
listing conventions in this book 250
properties, JavaScript 611
property-based event handlers 680
@property rule 595
protocols 24
pseudo-class selectors 332, 334–347
child-indexed 332
list of additional 347
location 334
logical 345–346
tree-structural 339–344
typed child-indexed 343
user action 334–336
pseudo-element selectors 348–350
pt unit
CSS 241
iOS reference pixel 508, 755
pull-down menus, forms 185
PWAs (progressive web apps) 714–720, 762
px unit 242, 259
Python scripting language 170

Q

q element 94
quality of images 789–790
quantization 738
queries
container 523–528
CSS feature queries 590–591
media 513–522
as RWD component 512
querySelector() 610, 637, 642, 656
querySelectorAll() 624
quotation element 82
quotes (")
strings 618, 619
wrapping attribute values in 66

R

radial gradients 321, 323–325
radio buttons, forms 183, 581
range input type, forms 190
raster images 134, 730
RDF Site Summary 801
React 695, 704, 705–706, 707
reader (user) style sheet 236
readonly attribute, forms 177
RealFaviconGenerator 765
Really Simple Syndication (RSS) 801
rect element, SVG 802
refactor, JavaScript 620
reference pixel 754
referrer attribute, iframe element 207
relative flex 451
relative font-sizing values (%, rem, em) 259–263
relative keywords, font size 262
relative positioning 411, 413–419
relative units, CSS 241, 242
relative URLs 116, 118–129, 137
rel attribute, link element 129, 763
removeChild() method 668
removeElement() method 667
rem unit 242, 259, 260, 514
rendering engines, browser 26
repeat() function, Grid Layout 473, 496
replaced element 136
required attribute, file input type 188
reserved characters, JavaScript 623
reset button, forms 180
resolution, image 752–753, 786
RespImageLint bookmarklet 787
Responsive Image Breakpoints Generator (Cloudinary) 788
responsive images
automating 794
image production 785–788
markup 149, 772, 786
Responsive Resources 549
Responsive Web Design (Marcotte) 549
responsive web design (RWD) 41–42, 505–550
choosing breakpoints 529–530
components of 507
content hierarchy 531–534
flexible images 510–512
fluid layouts 509–510
Grid Layout 494–500
images 133, 149, 511, 543, 772, 785–788
importance of 505–506
layout patterns 535–536
navigation 541–542
resources 548
setting viewport 507–508
special content 544
SVGs 831–832
testing 546–547
typography 537–540
videos 544
Retina screen 754
return keyword, JavaScript 642
revert property value 250
rgba() color function 305
RGB color model 293, 298–301
color picker 298–299
hexadecimal values 300–301
rgb() functional notation 299
transparency values 306
variations of 295
rgb() functional notation 299
rich-text documents 54
rights-managed 727
roles, ARIA 106
root directory 118
root element 59
:root selector 339
Rosenfeld, Louis 13
rotate() function, transforms 218, 561, 562, 567
rounded corners 377–378
routing, JavaScript 701
row-gap property 493–494
row group elements 162
row keyword, Flexbox 431
rows and columns, tables 156
rows attribute, textarea 176
rowspan attribute, table cells 159
royalty-free 727
RSS (Really Simple Syndication) 801
ruby annotation 100
Ruby on Rails platform 170
rule order, style rule 237
runtime, JavaScript 712
RWD. *See* responsive web design

S

Safari
animation inspector in 574
developer tools 246, 629
support for 588
turning on feature flags in 587
viewport on 508
Safari Technology Preview (Apple) 587
Salesforce, Lightning Design System 533
samp element 96
sandbox attribute, iframe element 207
sans-serif typefaces 228, 253
Sass (Syntactically Awesome Style Sheets) 589, 593
saturation, HSL color model 301
Save for Web feature (Photoshop) 741, 777
scale() function, transforms 561, 565–566
scale-independent pixel (sp) 755

scale property 836
scaling factor, line-height property 276
scaling SVGs 826–835, 836
scanning, as image source 726
scope
  custom properties 595
  JavaScript variables 640–641
scope attribute, table 161
:scope selector 334
screen density, grid 466
screen resolution 753–754
screen sizes, diversity challenge of 38
script element 710
scripting strategy, progressive enhancement 40
scripts, forms 167. *See also* JavaScript
scrolling menus, forms 185
scroll value, overflow property 366
search element 90
search engine optimization (SEO) 14, 775
search input type, forms 178
Section 508 guidelines 44
section element 87, 88, 114
sectioning elements 85–90
Secure Sockets Layer (SSL) protocol 27
security
  font file sources 253
  form control values check 179
  iframe concerns 207
  password entry field 177
  visited links 335
selected attribute, scrolling menu 186
select element 185–187, 541, 581
selectors 228, 229, 329–351
  attribute 330–332
  grouped 240
  pseudo-class. *See* pseudo-class selectors
  pseudo-element selectors 348–350
s element 93
self-end value, Grid Layout 492
self-hosted CDNs 795
self-hosted fonts 253
self-start value, Grid Layout 492
semantic markup 61, 75
semicolon (;)
  character references 107
  CSS declarations 229
  JavaScript 609
  statements 612
SEO (search engine optimization) 14, 775
SEO specialists 14
serif typefaces 252
serverless code 714
servers and server software 24–25
  backend development 12
  media format setup 210
  MIME type configuration 135
service worker 719
setAttribute() method 676, 685
SFC (Single File Component) 709
Shadow DOM specification 699
shorthand properties
  animation property 574
  background property 317–318
  border-color property 374–375
  border-image property 380
  border property 375
  border-radius property 377–378
  border-style property 371–373
  border-width property 373–374
  flex-flow property 434
  font property 268
  font-variant property 268
  gap property 493
  grid property 487–489
  list-style property 287
  margin property 381–382
  outline property 376
  padding property 367–370
  transition property 557
siblings, document tree 234
sidebars (aside element) 89
Silver, Adam 196
Simmons, Jen 501, 509, 588
simple attribute selector 330
simulator 547
Single File Component (SFC) 709
single-line text field, forms 175
single quotes ('), JavaScript strings 618
site-building services 4
site diagrams 6
Siteinspire 549
site root relative URLs 124, 312
Sitnik, Andrey 764
size attributes
  drop-down menus 185
  multiline text 176
  scrolling menus 185
Sketch 18, 789, 791, 820, 824
skew() function, transforms 561, 567
skewing 567
slash (/)
  versus backslash 58
  division operator 614
  Unix pathname convention 124
small caps (font-variant property) 266, 268
small element ("small print") 94
Smart Object 790
Smashing Magazine 247, 837
SMIL (Synchronized Multimedia Integration Language) 818
smoothing images 776
snake_case 623
Sneyers, Jon 749
soft skills 15
soft value, textarea 177
software, web development 16–21
Sorhus, Sindre 586
Sorrentino, John 765
sound state pseudo-class selectors 347
source document, viewing 30, 32, 162
source elements, embedded media 148, 212, 213
space-around property 436
space-between property 436
spam-bots 129
span element 103, 114
spanning table cells 158–159
span value, Grid Layout 481
specificity style rule 237, 272–273
split() function 637
spread, box-shadow property 389
sprites, SVG 811
sp (scale-independent pixel) 755
square brackets ([ ]), grid templates 478
Squarespace 4, 247
Squoosh 781
src attribute 65
  with audio files 214
  iframe element 206
  img element 136, 137, 676
  track element 215
  video 211, 212
src property, @font-face rule 256
srcset attribute
  img element 149
  source element 148
sRGB (Standard RGB) color space 294, 295, 298
SSL (Secure Sockets Layer) protocol 27
stacking order 422
Stack Overflow 721
standalone SVG 144
standards compliance 39, 699
start attribute, ol element 81, 114
start tag 58
StatCounter's Global Stats 531
statements, JavaScript 609, 612, 614

states and properties, ARIA 107
static positioning 411
static versus dynamic websites 34
step attribute, form controls 190
step-end keyword 555
steps keyword 555
step-start keyword 555
sticky headings 412
sticky positioning 412
stock photography and illustrations 727–729
Storey, Dudley 837
storyboards 6
streaming media 211
stretch keyword 490
strictly typed, JavaScript 703
strike-through, text 278
strings, JavaScript 611, 618–621
  concatenating 619–620
  containing quotes 619
  multiline 618
stroke, SVG 804
strong element 92, 114
structure (structural layer) 10, 33
  and CSS 233
  providing for simple HTML document 57–58
style attribute
  HTML elements 232, 272
  SVG 814
style element 230, 710
style guide 533
style inspector, developer tools 245
style rules 228–230
  handling conflicts (cascade) 235–239
  priority by origin 236
  priority by rule order 237
  priority by specificity 237
style sheets. *See* CSS
style tiles approach to visual design 9
styling strategy, progressive enhancement 40
Stylus 593
subdomains 28
sub element 96
subgrid 474
Sublime Text 17
submit button, forms 180, 196
subscript font-variant 268
subsequent-sibling selector (~) 271, 330
subtraction assignment operator (-=) 616
subtraction (-) operator 613
summary element, disclosure widget 197
SumoPaint 19
sup element 96
superscript font-variant 268
Super Zoom filter (Photoshop) 793
@supports rule 590
Svelte 704, 710
SVG 2 spec 802
SVG Artista 817
SVGator 817
SVG-Edit 821
svg element 144, 802–804, 826
SVGO 824
SVGOMG tool 825
SVG (Scalable Vector Graphics) 133, 731, 799–838
  adding to pages 141, 143–147, 813
  animation tools 817
  as background images 146
  versus bitmap images 143
  coordinates 804
  design tools 820
  drawing tools specific to 821
  elements 802–803
  embedded bitmap images 806
  filter effects 809
  img element embedding 144
  inline in HTML source 144–145
  interactivity with JavaScript 815–817
  for logos and icons 750, 771
  versus PNGs 763
  production tips 822–824
  in responsive layouts 785–787, 831
  scaling 826–835, 836
  styling 813–814
  symbol element 810–812
  and XML 801–811, 813–818
switch statement 647
symbol element, SVG 802, 811
Synchronized Multimedia Integration Language (SMIL) 818
synchronous code 706
Syntactically Awesome Style Sheets (Sass) 593
system fonts 267

T

table element 156
tables 153–166
  accessibility 159–161
  element list 166
  headers 157
  minimal structure 155–157
  row and column groups 161–162
  in RWD 545
  spanning cells 158–159
  styling 583
tabs, browser handling of 57
tab-size property 282
tags, HTML 30, 32, 58
  ignoring of unknown tags by browser 57
  omitting closing tags for block elements 76
Tailwind CSS 600
target attribute, opening page in new window 128
:target selector 334
:target-within selector 334
tbody element 162
td element 156, 157
technical text elements (code, var, samp, kbd) 96
telephone links 130
tel input type, forms 178
template element, in Vue 710
template literals, JavaScript 618
template properties, Grid Layout 469–470
template property, in Angular 709
template strings, JavaScript 618
Terminal application 21
testing
  in RWD 546–547
  for user research 6
text, adding large amounts to DOM 671
text-align-last property 277, 282
text-align property 277–278
textarea element, forms 175, 176, 580
text-autospacing property 282
textContent property, JavaScript 672
text-decoration property 278
TextEdit 54
text editors 52–54, 55–57
text elements 75–114. *See also* text-level (inline) elements
  accessibility with ARIA 105–107
  blockquote 82
  element list 114
  figure 84
  formatting. *See* formatting text with CSS
  generic (div, span) 101–105
  headings (h1 to h6) 76–77
  horizontal rule (hr) 78
  lists (ul, ol, li, dl, dt) 78
  paragraph (p) 75
  preformatted (pre) 83

sectioning elements 85–90
size of text properties 268
SVG 802
text entry form controls 175–179, 580
text-indent property 276
text-justify property 277, 282
text-level (inline) elements 63, 90–101
floating 398–399
margins on 384
text nodes, DOM 666, 673
text-shadow property 281–282, 389
text tracks, adding to video or audio 214–216
text-transform property 279
text wrapping
floating 396
with CSS shapes 407–408
tfoot element 162
thead element 162
th element 156, 157
theme toggle 664
third-party services
font hosting 254
for testing 547
three-box technique, style rules 272
3D transforms 570
Three.js 697
Thumbor 795
tilde (~), subsequent-sibling selector 271
tiling background images 313
time dimensional pseudo-class selectors 347
time element 97–98, 114
time selection, forms 188–189
timing functions 553–555
title attribute
with abbr element 95
JavaScript 685
title element 61
HTML 74
SVG 819
Topaz Labs' Gigapixel 794
top of the page, link back to 127
touch devices, hover limitation 336
track element, embedding content 214–216
tracks, grid
adding space between 493–494
aligning within grid containers 490–491
sizing 465
content-based 473
repeating 473
specifying values for 471–472
transformable elements 562
transform-origin property 563
transform property 561–562, 836
transforms 561–571
applying multiple 567–568
of position 564–565
properties (list) 578
reference box 565
rotation 562–564
size 565–567
skewing 567
transition-delay property 557
transition-duration property 553
transition-property property 553, 559
transitions 551–560
applying multiple 557
properties (list) 578
setting delays 557
shorthand transition property 557
timing functions 553–555
transition-timing-function property 554–556
translate() function, transforms 561, 564, 567
transparency 756–761
AVIF 748
for CSS color values 305
GIF 746
HSL 306
JPEG-XL 749
OKLCH 303
PNG 736, 738
RGB 306–307
WebP 743
transparent keyword, color values 296
tree-structural pseudo-classes 339–344
tr element 156
Trident rendering engine 26
triple equals sign (===), equality comparison 615
troubleshooting HTML source 69–70
Truecolor format 733
true/false values, JavaScript 649
truthy 649
tweening 552
24-bit color format 733
two dashes (--), custom properties 594
2D transforms 561
2x displays 787
type attribute
input element 174
link element 763
linking to external stylesheet 353
object element 208
source element 148
typed child pseudo-selectors 343–344
typeface 251
TypeScript 703
typography 268
fluid 260, 538–539
responsive 537–539

U

UCD (user-centered design) 5
u element 94
UI tool library 533
UI (user interface) design 8, 9, 18, 664–665
ul element 79–80, 117
Ultimate CSS Gradient Generator from Colorzilla 325
undefined variables 623
underlining text 278, 371
underscore (_), in variable names 623
Unicode (Universal Character Set) 59, 100
units of measurement, CSS 241–244
universal selector (*) 272, 330
unordered lists 79–80
UNPKG 695
unset property value 250
Unsplash 727
upscaling images 789, 791–793
url() function 311, 408
url input type, forms 178
URLs (Uniform Resource Locators) 26–29
absolute 116
relative 116, 118–129, 137
simplified 28–29
trailing slash purpose 29
use element, SVG 802, 811
user action pseudo-classes 334–336
user agent 25
user agent style sheets 64, 236, 245, 583
user-centered design (UCD) 5
user coordinate system, SVG 827
user experience (UX) design 5–7, 9, 196
user flow charts 7
user interface (UI) design 8, 9, 18, 664
user preference, media features 515
user space 826
user style sheet 236
user unit, SVG 803
UTF-8 encoding 59, 108
utility classes 600
UXPin 18

UX (user experience) design 5–7, 9, 196

V

V8 video codec 209
V9 video codec 209
validation, HTML documents 71–72
validator software 71
value attribute
  buttons in forms 180
  data element 98
  form menu options 185
  form text input 176
var() function 594, 595
variable declaration 621
variable fonts 255, 256
Variable Fonts Primer 255
variables
  CSS 592, 594–596
  declaring with let 624
  form 172
  JavaScript 621–625
  naming 623
  undefined 623
variable scope 595, 640–641
variable/value pairs, forms 173
variation axis, variable fonts 255
var keyword 624, 641
vector-based illustration tools 820
vector-based UI design tools 820
vector images 134, 789. *See also* SVG
Vectorizer 789
vendor prefixes 589
Verou, Lea 555
vertical-align property 282
vh unit 243
video
  adding text tracks 214–216
  deep linking 214
  iframe element 205
  in RWD 544
video element 209
  adding video to web page 211–213
  custom players 213
  providing format options 212
video files, encoding tools 210
video formats 209
Video.js 213
viewBox attribute, SVG 827
viewing a source document 30, 32, 162
viewports 25, 411
  features of 515
  percentage length units 243
  and RWD 41, 507
  SVG 803, 804, 826–828
  width dimensions 518
virtual devices for testing 546
virtual DOM 706
virtual machine (VM) software 16
visible value, overflow property 366
vision impairment 43
visited links 334, 335
:visited selector 334
visual designers 5, 8
Visual Studio Code 17
vmax unit 244
vmin unit 244
VM (virtual machine) software 16
void elements 32, 64, 65, 78, 136
Vorbis audio codec 210
Vue 709
vw unit 243

W

W3C. *See* World Wide Web Consortium
W3Schools 72
WAI-ARIA 44, 106
WAI (Web Accessibility Initiative) 43, 105
Walkway 817
Walton, Philip 527
Warren, Samantha 9
waterfall charts 46
Waterhouse, Tom 572
WAV audio format 210
wbr element 100
Web Accessibility Initiative (WAI) 43, 105
WebAIM 819
web app manifest 716–718
web browsers. *See* browsers
web components 698–700
Web Content Accessibility Guidelines (WCAG) 43
web design
  device multitude challenge 38–39
  how the web works 23–36
  roles and responsibilities 3–15
  standards compliance 39
  tools of the trade 15–21
Web.dev (Chrome Developer Relations) 72
web developers, what employers look for 12
web fonts, embedded 251, 253, 257
web forms. *See* forms
Web Hypertext Application Technology Working Group (WHATWG) 11
web images. *See* images
WebKit rendering engine 26
WebM
  audio format 210
  video format 209
web manifest 765
web pages 30–34
  addresses. *See* URLs
  creating basic elements of 51–74
  Flexbox for layout of 453–454
  Grid Layout for 469
  and HTML documents 30–32
Web Platform Baseline 588
WebRTC 211
web storage 689
weight of style, and position in cascade 235
Weizenbaum, Nathan 593
Weyl, Estelle 243, 501, 572
WHATWG (Web Hypertext Application Technology Working Group) 11
:where() selector 346
white-space property 282
widgets (forms) 167
width attribute
  canvas element 218
  iframe element 206
  img element 139, 140
  object element 208
  video element 211
width property
  absolute positioning 416
  box model 360, 362
  for floated text elements 399
  for floating block elements 401
  for inline elements 362
  svg element 803, 826, 830
width (w) descriptor 786
Wikimedia Commons 728
Willison, Simon 694
window-in-a-window viewing, iframe 205–207
Windows environment 16
  on MacOS 16
  server software in 24
  text editor 53–54
wireframe diagrams 6
Wix 4
WOFF2 (Web Open Font Format 2) 253
word-boundary-detection property 282
word-break property 282
WordPress 4, 781, 796

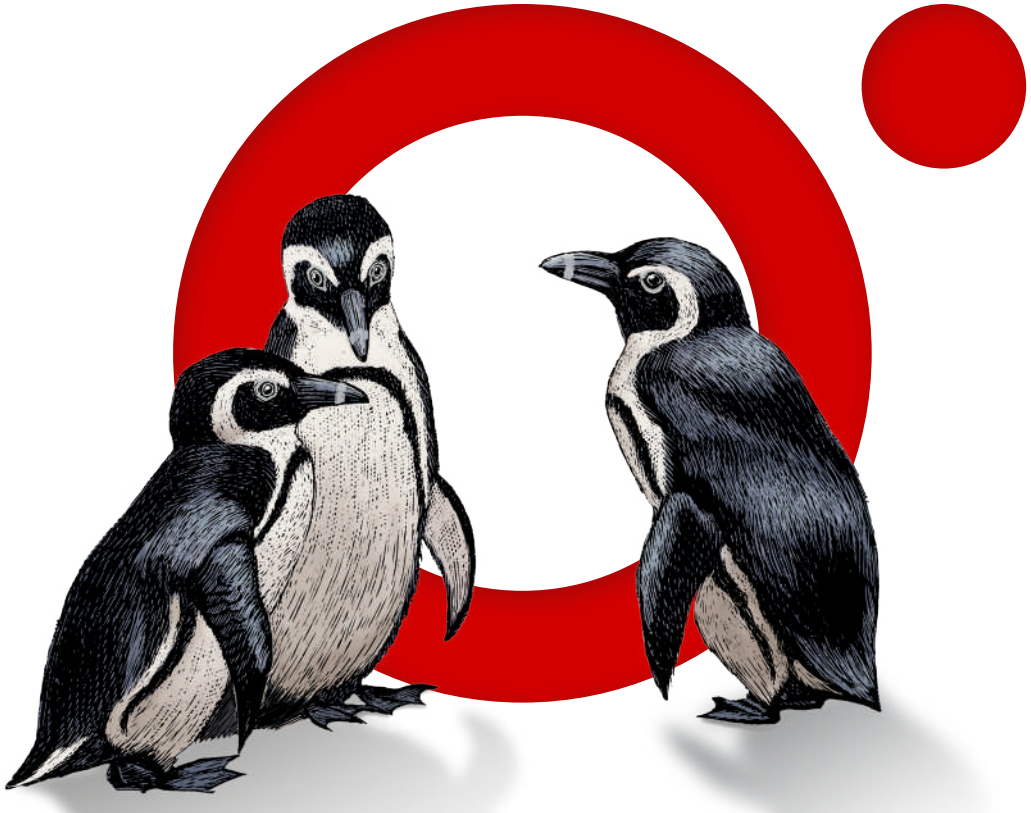
word-spacing property 280
World Wide Web Consortium (W3C) 11, 246
  history 24
  standards development 39
  and WAI 43
World Wide Web (WWW) 24
wrap attribute, textarea 177
writing code 10–13
Wroblewski, Luke 196, 535

X

X11 color names 296
x (density) descriptor 786
x-height 268
XHTML 58, 801
XML Character Data block 816
XML (Extensible Markup Language) 141, 801
xmlns attribute, SVG 803
XSLT (eXtensible Stylesheet Language Transformations) 819

Z

Zakas, Nicholas 720
z-axis 422
z-index property 422



O'REILLY®

Learn from experts. Become one yourself.

60,000+ titles | Live events with experts | Role-based courses
Interactive learning | Certification preparation

Try the O'Reilly learning platform free for 10 days.

©2025 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly Media, Inc. 718900_8x9.75